How to check mvn is installed in my computer or not

mvn -version

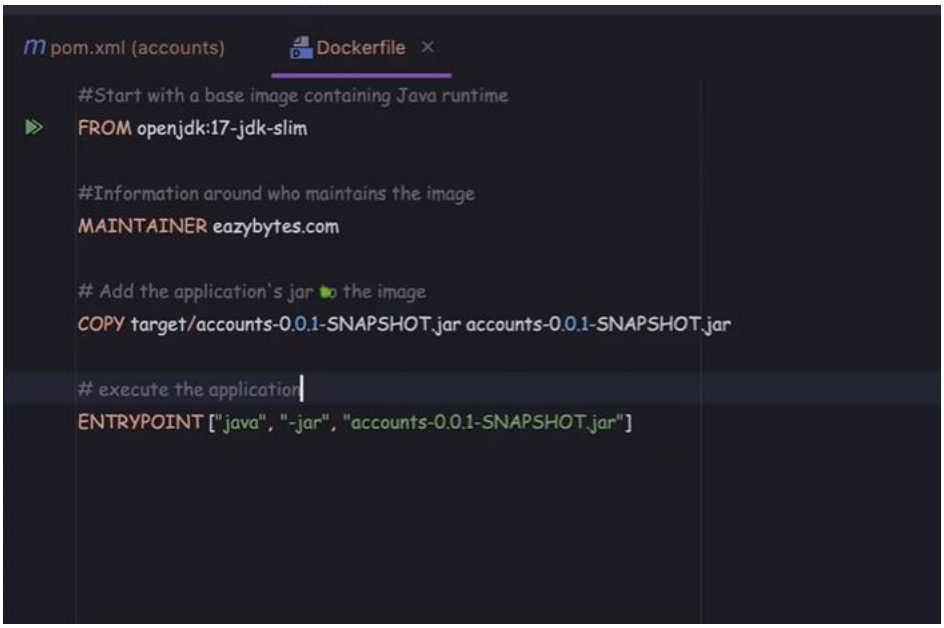
How to check docker is installed or not

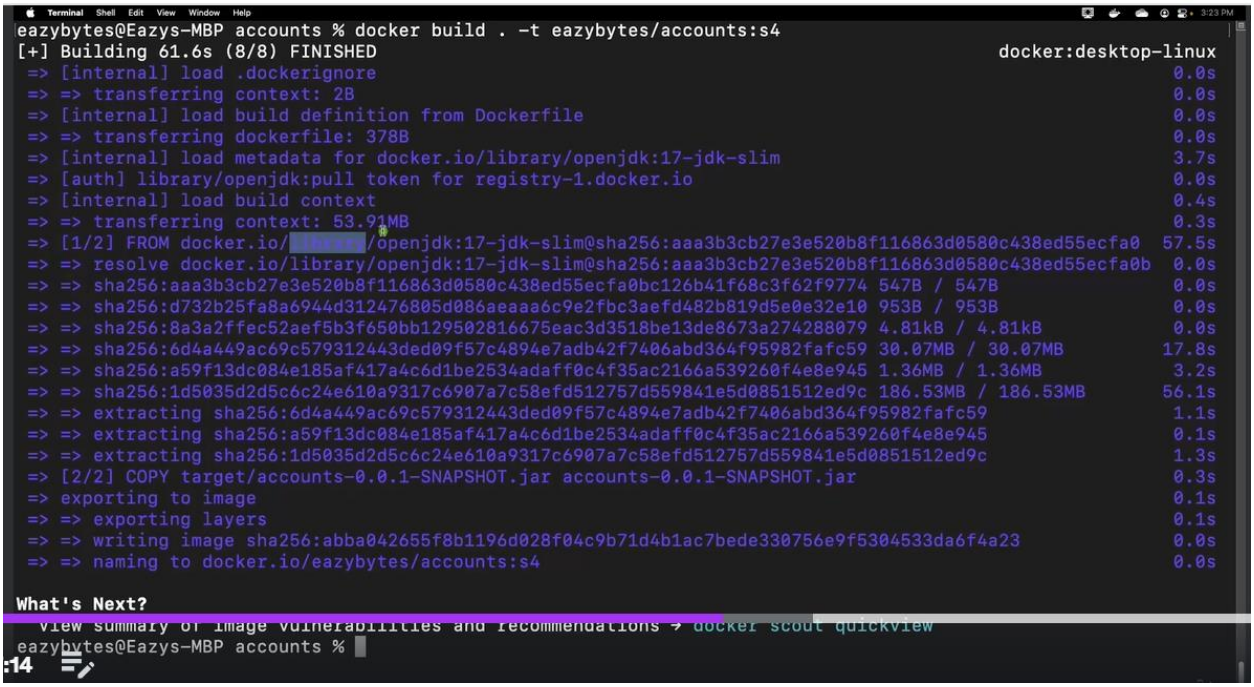
docker -version

Command to generate jar from spring boot project jar

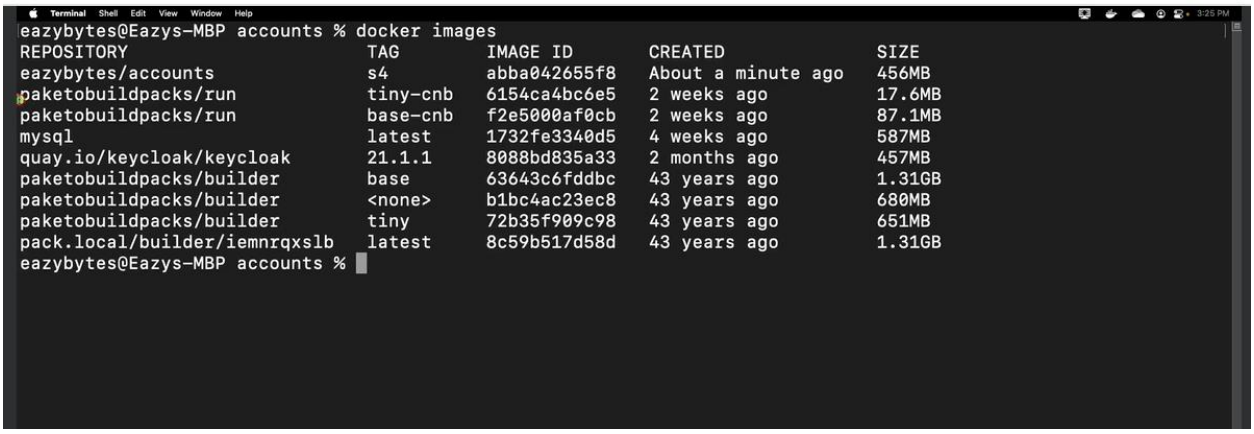
- ⇨ mvn clean install generate jar file inside target folder
- ⇨ mvn spring-boot:run to run the spring boot application
- ⇨ java -jar target/account

Create Docker File

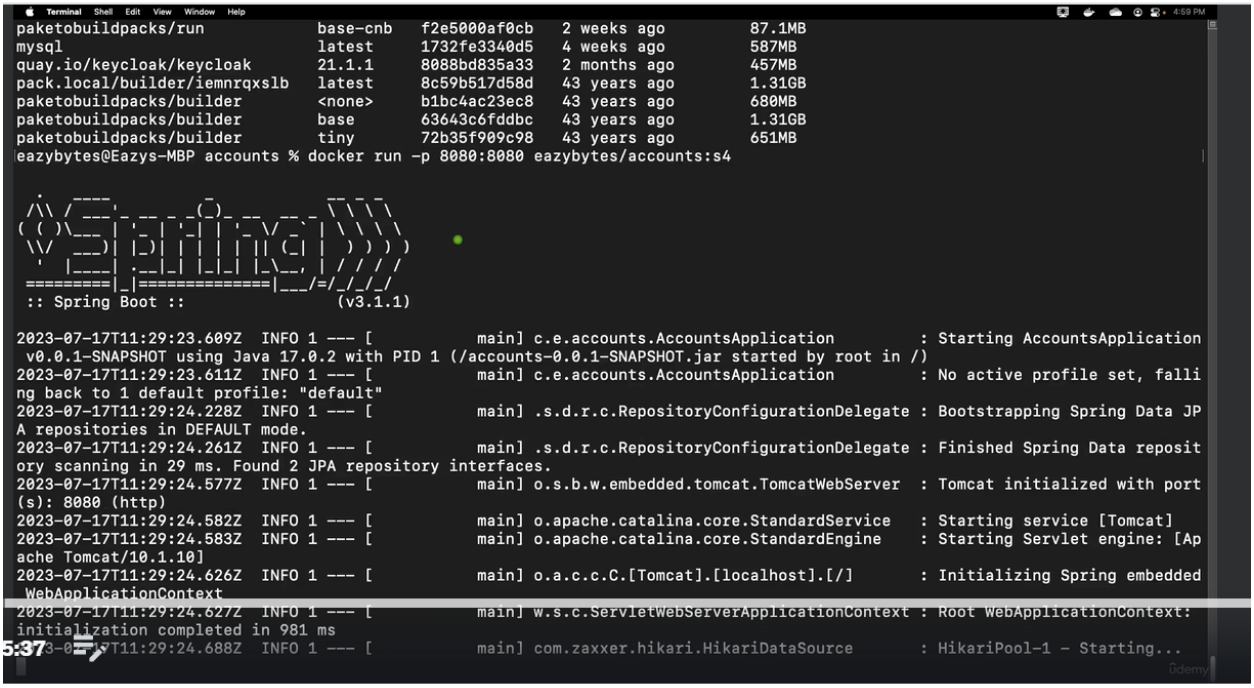
1. create a file inside the application with no extension name as Dockerfile
2. 



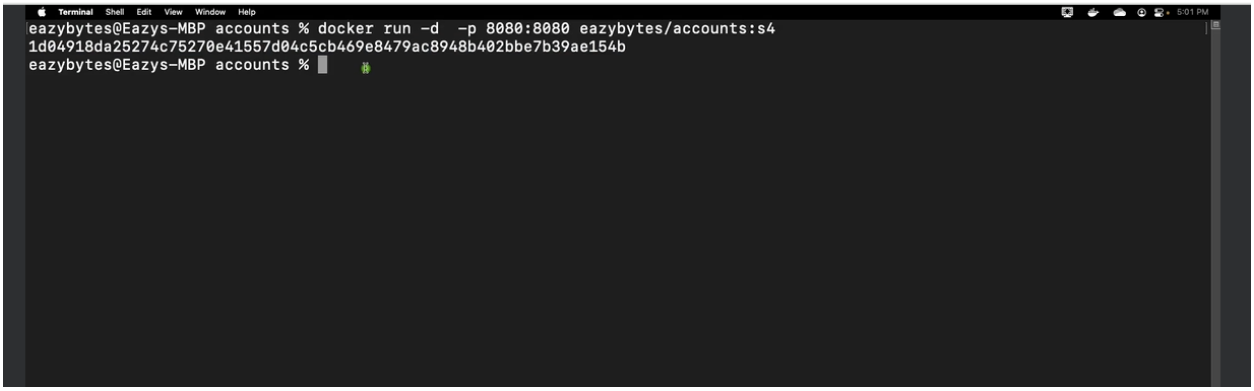
Command to search Docker image in side docker local server using docker cli



How to Run docker container from a docker image



Running docker container in detached mode



Here docker container is started as detached mode and we can use other command also in my docker cli to do other work for docker

How to check existence running container

```
eazybytes@Eazys-MBP accounts % docker run -d  -p 8080:8080 eazybytes/accounts:s4
1d04918da25274c75270e41557d04c5cb469e8479ac8948b402bbe7b39ae154b
eazybytes@Eazys-MBP accounts % docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS                    NAME
S
1d04918da252   eazybytes/accounts:s4   "java -jar accounts-…"   23 seconds ago   Up 22 seconds   0.0.0.0:8080->8080/tcp   stra
nge_mcclintock
eazybytes@Eazys-MBP accounts %
```

Docker command to see all stopped container in docker servers

```
eazybytes@Eazys-MBP accounts % docker run -d  -p 8080:8080 eazybytes/accounts:s4
1d04918da25274c75270e41557d04c5cb469e8479ac8948b402bbe7b39ae154b
eazybytes@Eazys-MBP accounts % docker ps
CONTAINER ID   IMAGE                  COMMAND                  CREATED          STATUS          PORTS                    NAME
S
1d04918da252   eazybytes/accounts:s4   "java -jar accounts-…"   23 seconds ago   Up 22 seconds   0.0.0.0:8080->8080/tcp   stra
nge_mcclintock
eazybytes@Eazys-MBP accounts % docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
eazybytes@Eazys-MBP accounts % docker ps -a
CONTAINER ID   IMAGE                                  COMMAND                  CREATED              STATUS
       PORTS     NAMES
1d04918da252   eazybytes/accounts:s4                  "java -jar accounts-…"   About a minute ago   Exited (143) 34 seconds ag
o                strange_mcclintock
83e66dacbaa7   eazybytes/accounts:s4                  "java -jar accounts-…"   3 minutes ago        Exited (130) 2 minutes ago
                 youthful_fermat
d44fa4a36a13   eazybytes/accounts:s4                  "java -jar accounts-…"   About an hour ago    Exited (130) About an hour
 ago             epic_mclean
96128de15fe2   pack.local/builder/iemnrqxslb:latest   "/cnb/lifecycle/crea…"   5 days ago           Exited (0) 5 days ago
                 exciting_vaughan
59ed928c8f65   mysql                                  "docker-entrypoint.s…"   2 weeks ago          Exited (0) 2 weeks ago
                 cardsdb
9cd4cdb04ea5   mysql                                  "docker-entrypoint.s…"   2 weeks ago          Exited (0) 2 weeks ago
                 loansdb
b307e66a83f4   mysql                                  "docker-entrypoint.s…"   2 weeks ago          Exited (0) 2 weeks ago
                 accountsdb
37dfe6e2d97b   quay.io/keycloak/keycloak:21.1.1       "/opt/keycloak/bin/k…"   2 weeks ago          Exited (143) 2 weeks ago
                 clever_chaum
eazybytes@Eazys-MBP accounts %
```

PORT MAPPING IN DOCKER

eazy bytes

What is port mapping or port forwarding or port publishing ?

By default, containers are connected to an isolated network within the Docker host. To access a container from your local network, you need to configure port mapping explicitly. For instance, when running the accounts Service application, we can provide the port mapping as an argument in the docker run command: **-p 8081:8080** (where the first value represents the external port and the second value represents the container port). Below diagram demonstrates the functionality of this configuration.

:37

Running a Spring Boot app as a container using Dockerfile

eazy bytes

STEPS TO BE FOLLOWED

1) Run the maven command, "mvn clean install" from the location where pom.xml is present to generate a fat jar inside target folder

2) Write instructions to Docker inside a file with the name Dockerfile to generate a Docker image. Sample instructions are mentioned on the left hand side

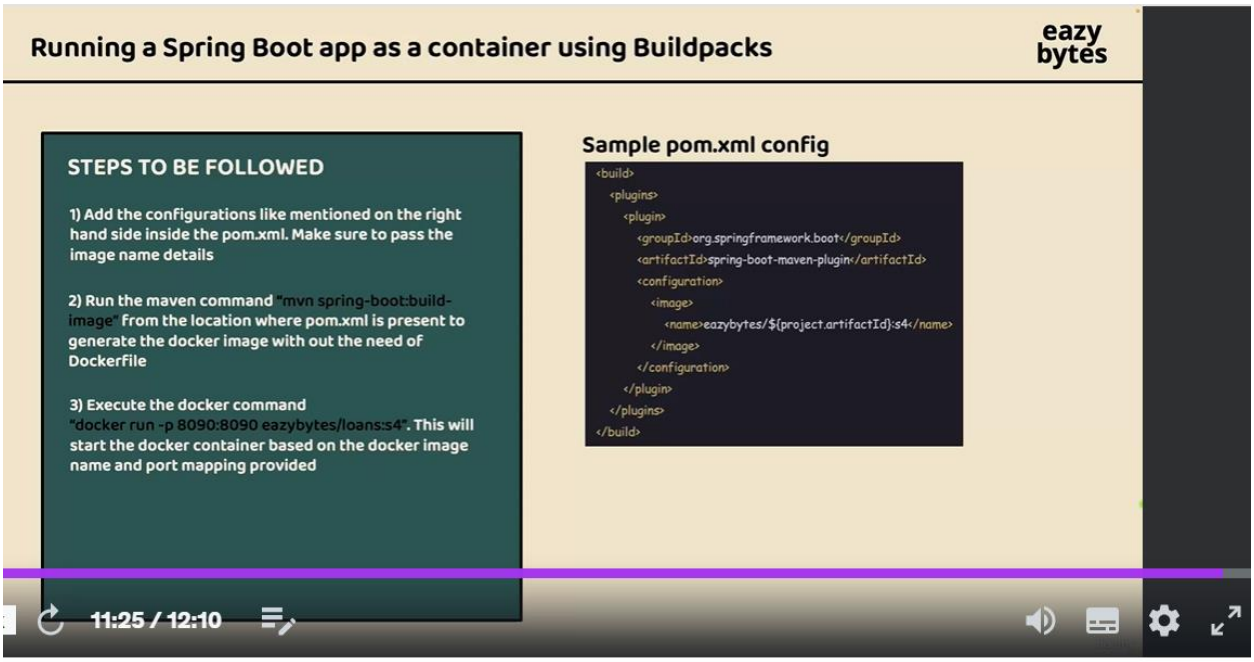
3) Execute the docker command "docker build . -t eazybytes/accounts:s4" from the location where Dockerfile is present. This will generate the docker image based on the tag name provided

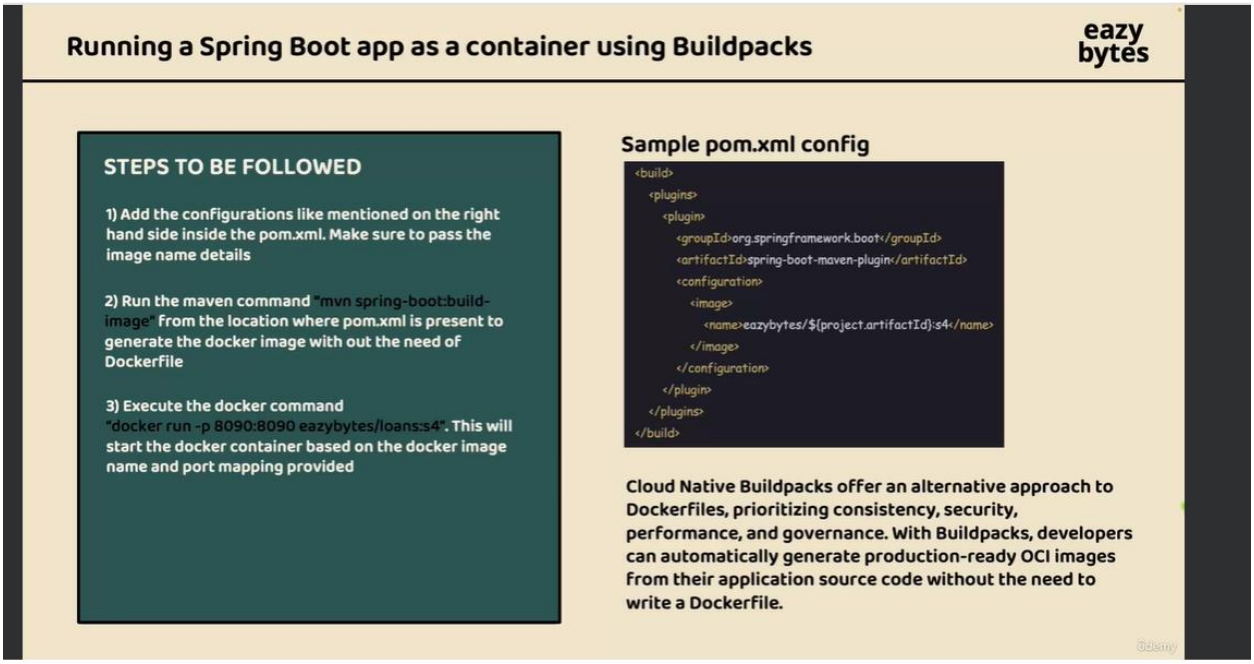
4) Execute the docker command "docker run -p 8080:8080 eazybytes/accounts:s4". This will start the docker container based on the docker image name and port mapping provided

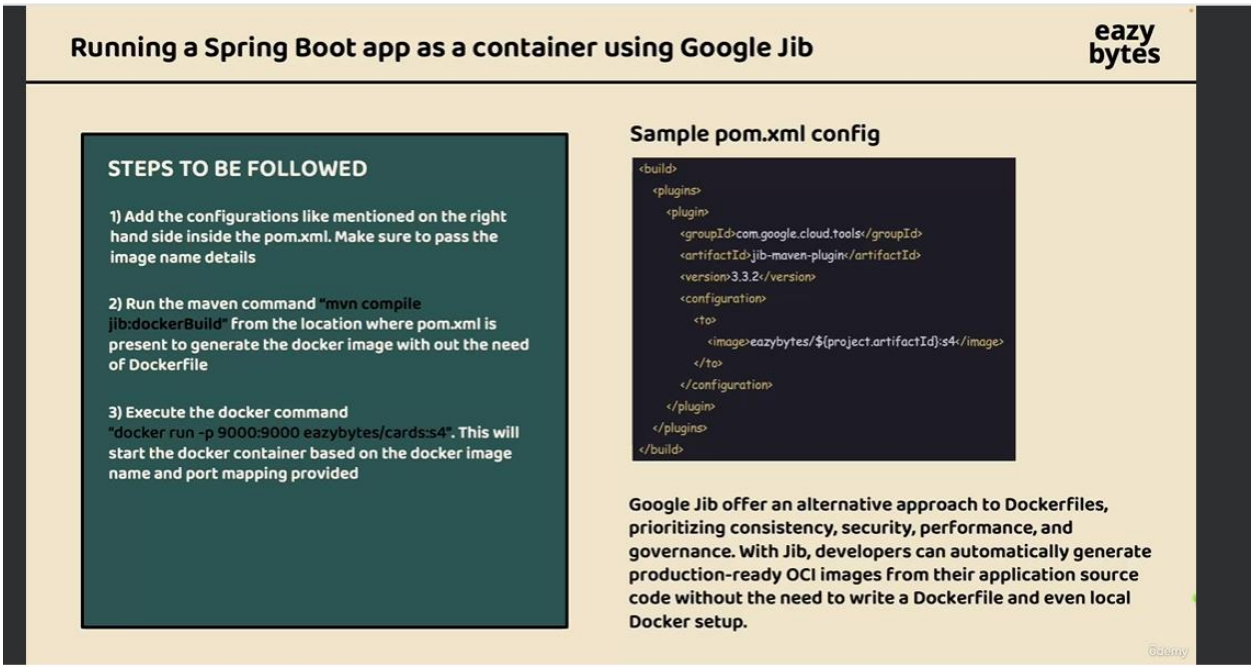
Sample Dockerfile

```
#Start with a base image containing Java runtime
FROM openjdk:17-jdk-slim

#Information around who maintains the image
MAINTAINER eazybytes.com

# Add the application's jar to the container
COPY target/accounts-0.0.1-SNAPSHOT.jar accounts-0.0.1-SNAPSHOT.jar

#execute the application
ENTRYPOINT ["java","-jar","/accounts-0.0.1-SNAPSHOT.jar"]
```

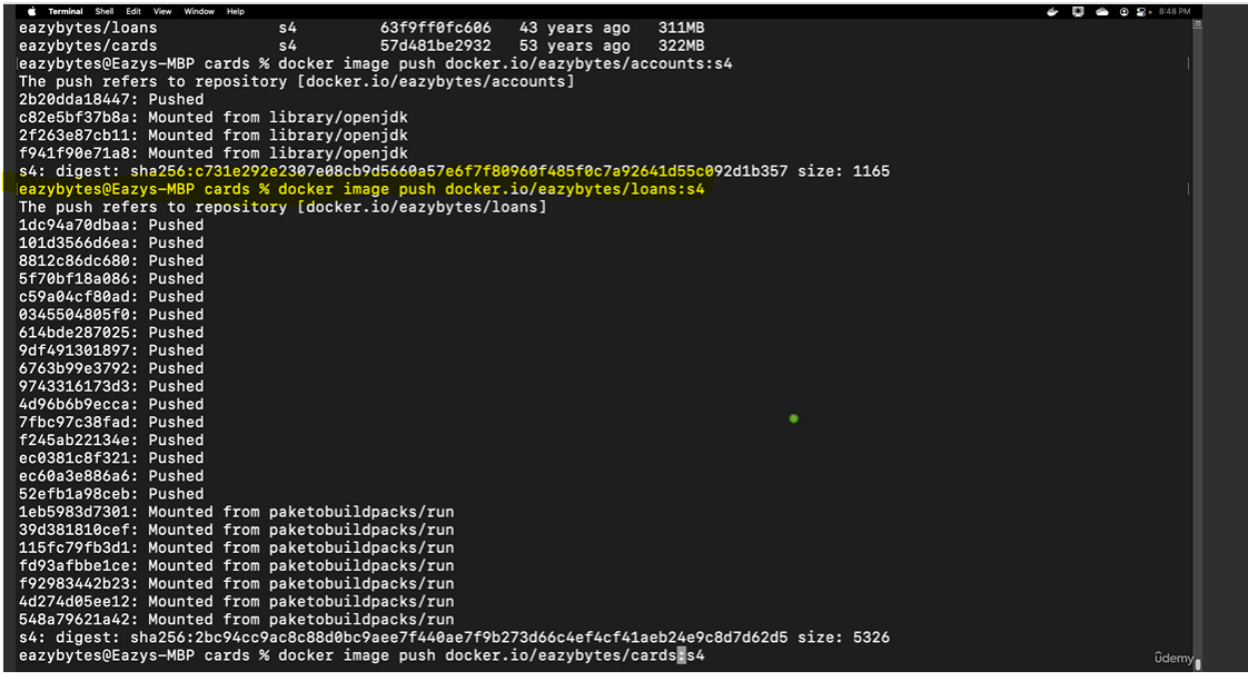
5x 15:26 / 15:37


Running a Spring Boot app as a container using Buildpacks
eazy bytes
STEPS TO BE FOLLOWED
1) Add the configurations like mentioned on the right hand side inside the pom.xml. Make sure to pass the image name details
2) Run the maven command "mvn spring-boot:build-image" from the location where pom.xml is present to generate the docker image with out the need of Dockerfile
3) Execute the docker command "docker run -p 8090:8090 eazybytes/loans:s4". This will start the docker container based on the docker image name and port mapping provided
Sample pom.xml config
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<image>
<name>eazybytes/${project.artifactId}:s4</name>
</image>
</configuration>
</plugin>
</plugins>
</build>
11:25 / 12:10

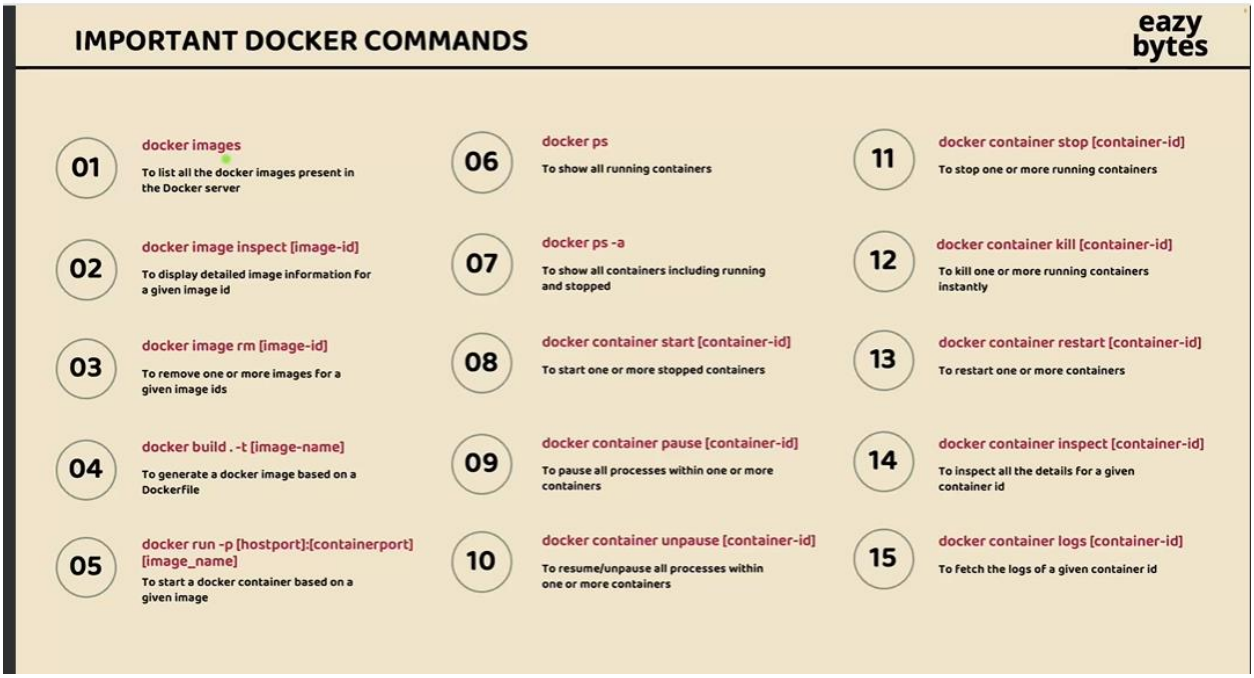

Running a Spring Boot app as a container using Buildpacks
eazy bytes
STEPS TO BE FOLLOWED
1) Add the configurations like mentioned on the right hand side inside the pom.xml. Make sure to pass the image name details
2) Run the maven command "mvn spring-boot:build-image" from the location where pom.xml is present to generate the docker image with out the need of Dockerfile
3) Execute the docker command "docker run -p 8090:8090 eazybytes/loans:s4". This will start the docker container based on the docker image name and port mapping provided
Sample pom.xml config
<build>
<plugins>
<plugin>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-maven-plugin</artifactId>
<configuration>
<image>
<name>eazybytes/${project.artifactId}:s4</name>
</image>
</configuration>
</plugin>
</plugins>
</build>
Cloud Native Buildpacks offer an alternative approach to Dockerfiles, prioritizing consistency, security, performance, and governance. With Buildpacks, developers can automatically generate production-ready OCI images from their application source code without the need to write a Dockerfile.



Push docker image from local to docker hub




Terminal Shell Edit View Window Help
9:34 PM
eazybytes@Eazys-MBP accounts % docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
0090fbe470f3 eazybytes/cards:s4 "java -cp @/app/jib-…" 14 seconds ago Up 13 seconds 0.0.0.0:9000->
9000/tcp cards-ms
3d3d6934b807 eazybytes/loans:s4 "/cnb/process/web" 14 seconds ago Up 13 seconds 0.0.0.0:8090->
8090/tcp loans-ms
2771c76f475f eazybytes/accounts:s4 "java -jar accounts-…" 14 seconds ago Up 13 seconds 0.0.0.0:8080->
8080/tcp accounts-ms
eazybytes@Eazys-MBP accounts % docker compose down -d
unknown shorthand flag: 'd' in -d
eazybytes@Eazys-MBP accounts % docker compose down
[+] Running 4/3
✔ Container accounts-ms Removed 0.3s
✔ Container loans-ms Removed 0.4s
✔ Container cards-ms Removed 0.4s
✔ Network accounts_eazybank Removed 0.1s
eazybytes@Eazys-MBP accounts %


IMPORTANT DOCKER COMMANDS
eazy bytes
01
docker images
To list all the docker images present in the Docker server
02
docker image inspect [image-id]
To display detailed image information for a given image id
03
docker image rm [image-id]
To remove one or more images for a given image ids
04
docker build . -t [image-name]
To generate a docker image based on a Dockerfile
05
docker run -p [hostport]:[containerport] [image_name]
To start a docker container based on a given image
06
docker ps
To show all running containers
07
docker ps -a
To show all containers including running and stopped
08
docker container start [container-id]
To start one or more stopped containers
09
docker container pause [container-id]
To pause all processes within one or more containers
10
docker container unpause [container-id]
To resume/unpause all processes within one or more containers
11
docker container stop [container-id]
To stop one or more running containers
12
docker container kill [container-id]
To kill one or more running containers instantly
13
docker container restart [container-id]
To restart one or more containers
14
docker container inspect [container-id]
To inspect all the details for a given container id
15
docker container logs [container-id]
To fetch the logs of a given container id

IMPORTANT DOCKER COMMANDS

eazy bytes

16 **docker container logs -f [container-id]**
To follow log output of a given container id

17 **docker rm [container-id]**
To remove one or more containers based on container ids

18 **docker container prune**
To remove all stopped containers

19 **docker image push [container_registry/username:tag]**
To push an image from a container registry

20 **docker image pull [container_registry/username:tag]**
To pull an image from a container registry

21 **docker image prune**
To remove all unused images

22 **docker container stats**
To show all containers statistics like CPU, memory, I/O usage

23 **Docker system prune**
Remove stopped containers, dangling images, and unused networks, volumes, and cache

24 **docker rmi [image-id]**
To remove one or more images based on image ids

25 **docker login –u [username]**
To login in to docker hub container registry

26 **docker logout**
To login out from docker hub container registry

27 **docker history [image-name]**
Displays the intermediate layers and commands that were executed when building the image

28 **docker exec -it [container-id] sh**
To open a shell inside a running container and execute commands

29 **docker compose up**
To create and start containers based on given docker compose file

30 **docker compose down**
To stop and remove containers for services defined in the Compose file

What are cloud native applications ?

eazy bytes

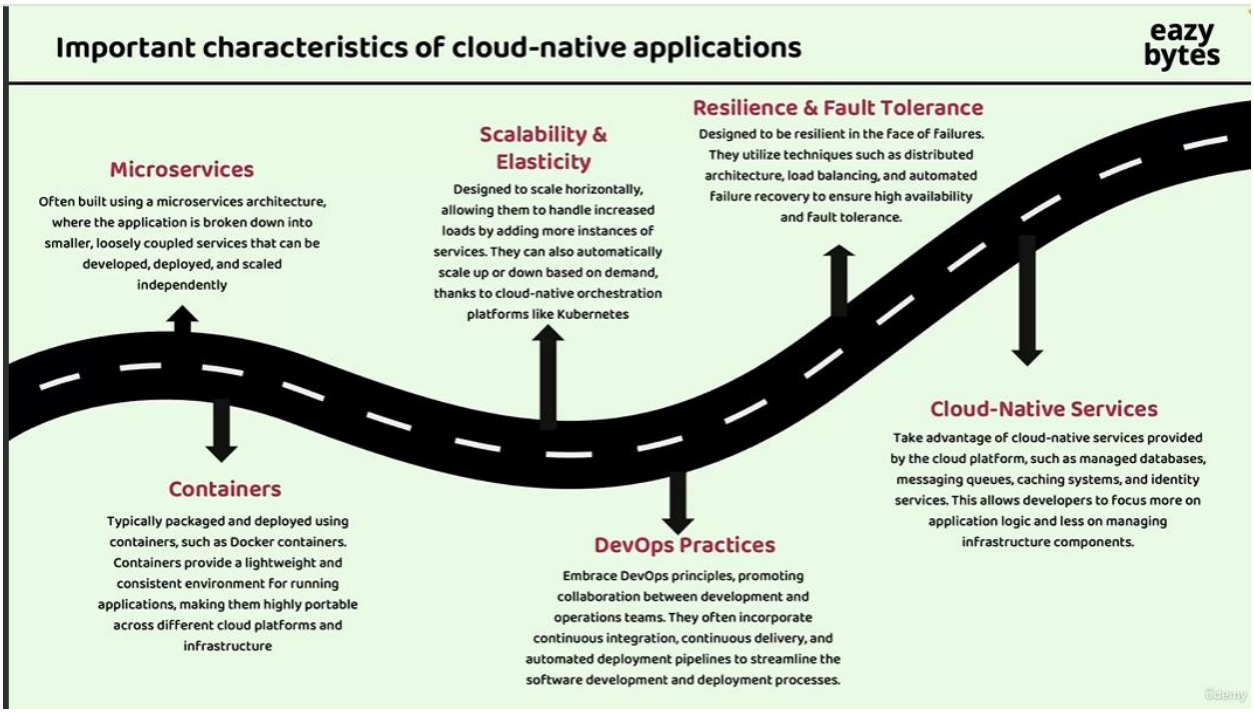
The layman definition

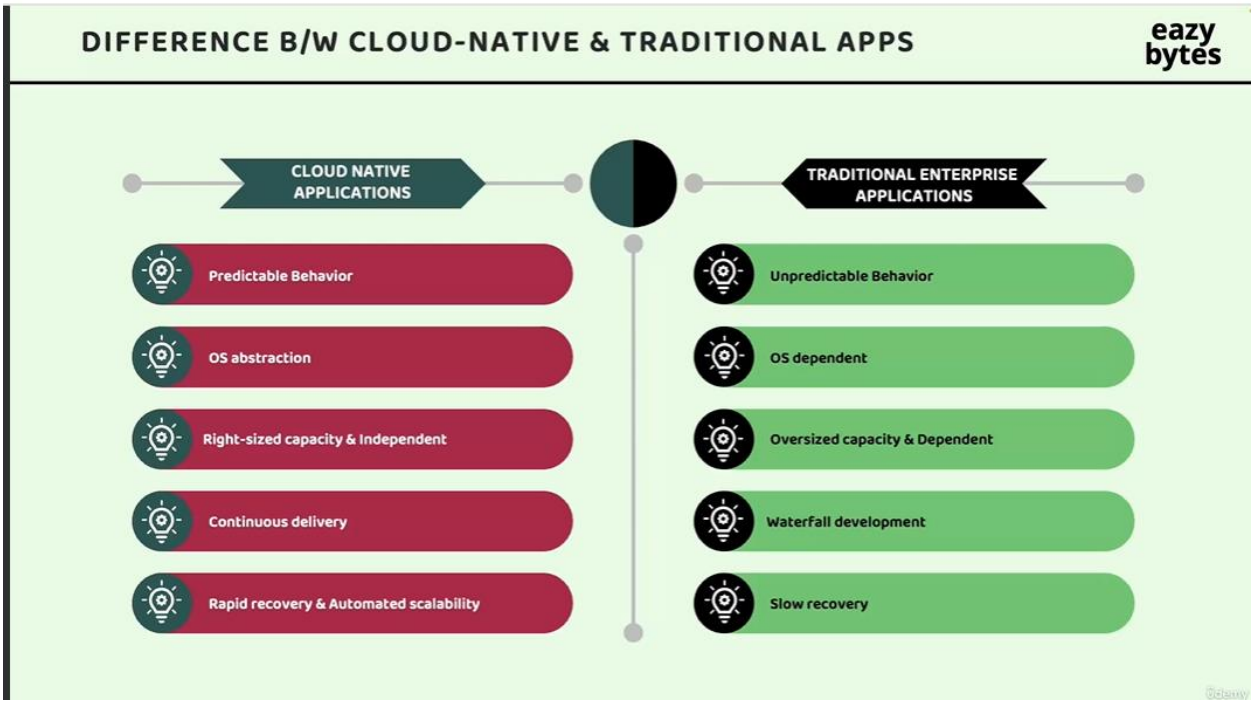
Cloud-native applications are software applications designed specifically to leverage cloud computing principles and take full advantage of cloud-native technologies and services. These applications are built and optimized to run in cloud environments, utilizing the scalability, elasticity, and flexibility offered by the cloud.

The Cloud Native Computing Foundation (CNCF) definition

Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach.

These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.


Important characteristics of cloud-native applications
eazy bytes
Microservices
Often built using a microservices architecture, where the application is broken down into smaller, loosely coupled services that can be developed, deployed, and scaled independently
Scalability & Elasticity
Designed to scale horizontally, allowing them to handle increased loads by adding more instances of services. They can also automatically scale up or down based on demand, thanks to cloud-native orchestration platforms like Kubernetes
Resilience & Fault Tolerance
Designed to be resilient in the face of failures. They utilize techniques such as distributed architecture, load balancing, and automated failure recovery to ensure high availability and fault tolerance.
Containers
Typically packaged and deployed using containers, such as Docker containers. Containers provide a lightweight and consistent environment for running applications, making them highly portable across different cloud platforms and infrastructure
DevOps Practices
Embrace DevOps principles, promoting collaboration between development and operations teams. They often incorporate continuous integration, continuous delivery, and automated deployment pipelines to streamline the software development and deployment processes.
Cloud-Native Services
Take advantage of cloud-native services provided by the cloud platform, such as managed databases, messaging queues, caching systems, and identity services. This allows developers to focus more on application logic and less on managing infrastructure components.


DIFFERENCE B/W CLOUD-NATIVE & TRADITIONAL APPS
eazy bytes
CLOUD NATIVE APPLICATIONS
Predictable Behavior
OS abstraction
Right-sized capacity & Independent
Continuous delivery
Rapid recovery & Automated scalability
TRADITIONAL ENTERPRISE APPLICATIONS
Unpredictable Behavior
OS dependent
Oversized capacity & Dependent
Waterfall development
Slow recovery

Development Principles of Cloud Native: 12 Factors & Beyond

eazy bytes

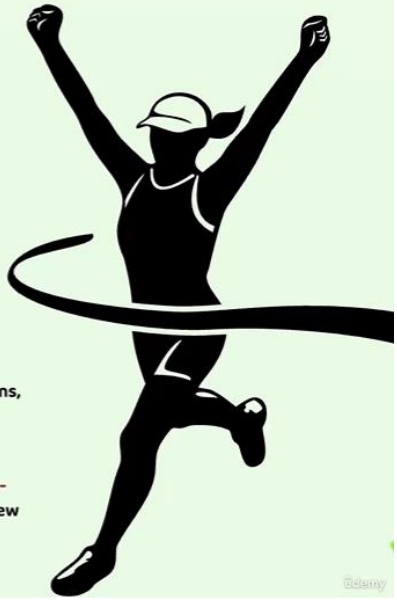
How to get succeeded in building Cloud Native Apps & what are the guiding principles that can be considered for the same ?

The engineering team at **Heroku** cloud platform introduced the **12-Factor methodology**, a set of development principles aimed at guiding the design and construction of cloud-native applications. These principles are the result of their expertise and provide valuable insights for building web applications with specific characteristics:

1) **Cloud Platform Deployment:** Applications designed to be seamlessly deployed on various cloud platforms.
2) **Scalability as a Core Attribute:** Architectures that inherently support scalability.
3) **System Portability:** Applications that can run across different systems and environments.
4) **Enabling Continuous Deployment and Agility:** Facilitating rapid and agile development cycles.

These principles were developed to assist developers in building effective cloud-native applications, emphasizing the key factors that should be considered for optimal outcomes.

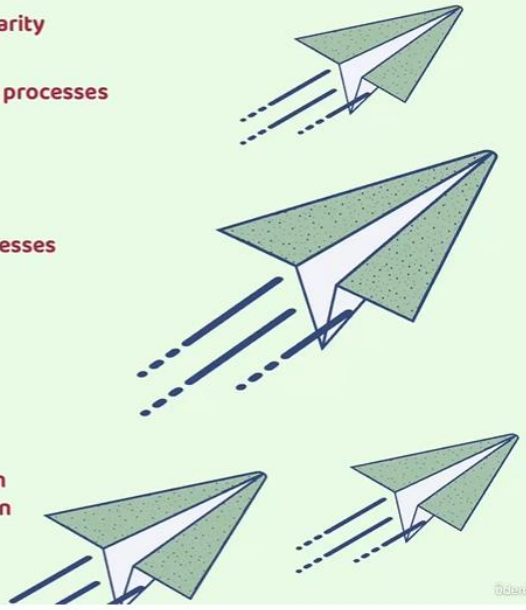
Subsequently, **Kevin Hoffman** expanded upon the original factors and introduced additional ones in his book, **"Beyond the Twelve-Factor App"** This revised approach, referred to as the **15-Factor methodology**, refreshing the content of the original principles and incorporates three new factors.



15-Factor methodology

eazy bytes

01 One codebase, one application
02 API first
03 Dependency management
04 Design, build, release, run
05 Configuration, credentials & code
06 Logs
07 Disposability
08 Backing services
09 Environment parity
10 Administrative processes
11 Port binding
12 Stateless processes
13 Concurrency
14 Telemetry
15 Authentication & authorization

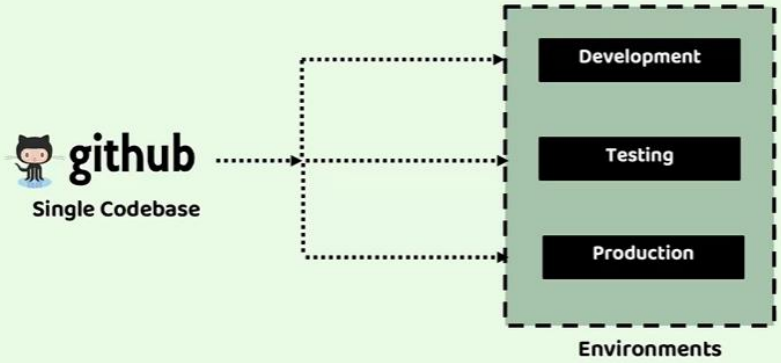


15-Factor methodology - One codebase, one application

eazy bytes

The 15-Factor methodology ensures a one-to-one correspondence between an application and its codebase, meaning each application has a dedicated codebase. Shared code is managed separately as a library, allowing it to be utilized as a dependency or as a standalone service, serving as a backing service for other applications. It is possible to track each codebase in its own repository, providing flexibility and organization.

In this methodology, a deployment refers to an operational instance of the application. Multiple deployments can exist across different environments, all leveraging the same application artifact. It is unnecessary to rebuild the codebase for each environment-specific deployment. Instead, any factors that vary between deployments, such as configuration settings, should be maintained externally from the application codebase.



15-Factor methodology - API first

eazy bytes

In a cloud-native ecosystem, a typical setup consists of various services that interact through APIs. Adopting an API-first approach during the design phase of a cloud-native application encourages a mindset aligned with distributed systems and promotes the division of work among multiple teams. Designing the API as a priority allows other teams to build their solutions based on that API when using the application as a backing service

This upfront design of the API contract results in more reliable and testable integration with other systems as part of the deployment pipeline. Moreover, internal modifications to the API implementation can be made without impacting other applications or teams that rely on it

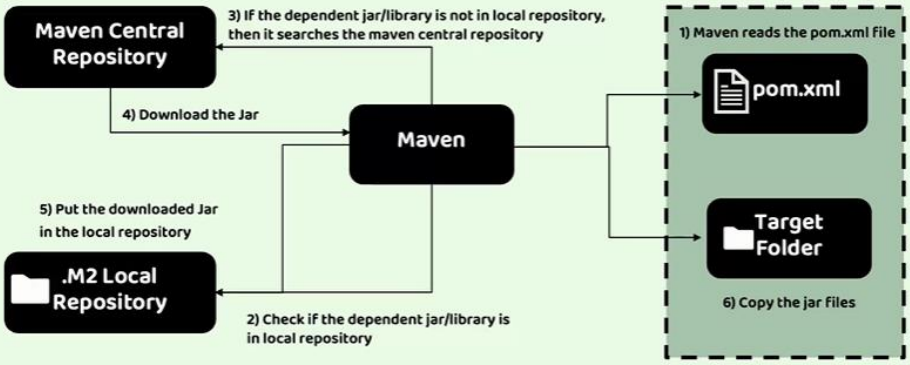
15-Factor methodology - Dependency management

eazy bytes

It is crucial to explicitly declare all dependencies of an application in a manifest and ensure that they are accessible to the dependency manager, which can download them from a central repository.

In the case of Java applications, we are fortunate to have robust tools like Maven or Gradle that facilitate adherence to this principle. The application should only have implicit dependencies on the language runtime and the dependency manager tool, while all private dependencies must be resolved through the dependency manager itself. By following this approach, we maintain a clear and controlled dependency management process for our application.

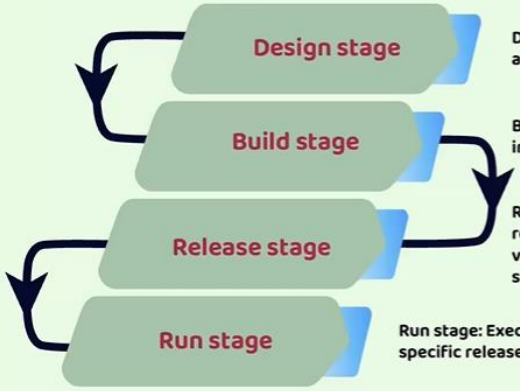
Sample flow when we use Maven as build tool



15-Factor methodology - Design, build, release, run

eazy bytes

Codebase progression from design to production deployment involves below stages,



Design stage: Determine technologies, dependencies, and tools for specific application features.

Build stage: Compile and package the codebase with dependencies, creating an immutable artifact (build). Unique identification of the build artifact is essential.

Release stage: Combine the build with a specific deployment configuration. Each release is immutable and uniquely identifiable, such as through semantic versioning (e.g., 6.1.5) or timestamp (e.g., 2023-08-15_12:01). Central repository storage facilitates easy access, including rollbacks if needed.

Run stage: Execute the application in the designated runtime environment using a specific release.

Following the 15-Factor methodology, these stages must maintain strict separation, and runtime code modifications are prohibited to prevent mismatches with the build stage. Immutable build and release artifacts should bear unique identifiers, ensuring reproducibility.

15-Factor methodology - Configuration, credentials & code

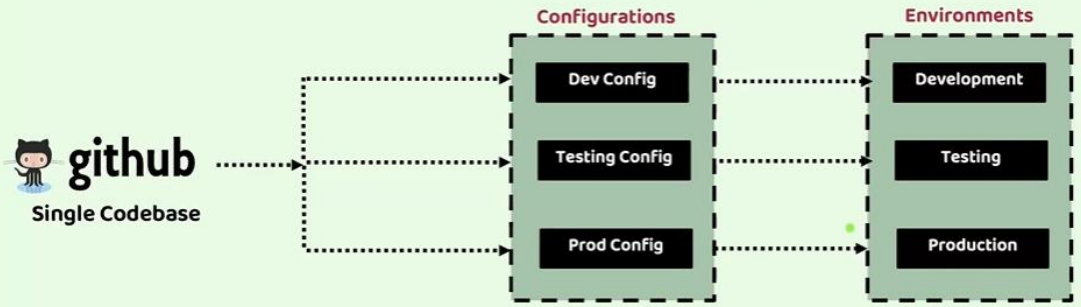
eazy bytes

According to the 15-Factor methodology, configuration encompasses all elements prone to change between deployments. It emphasizes the ability to modify application configuration independently, without code changes or the need to rebuild the application.

Configuration may include resource handles for backing services (e.g., databases, messaging systems), credentials for accessing third-party APIs, and feature flags. It is essential to evaluate whether any confidential or environment-specific information would be at risk if the codebase were exposed publicly. This assessment ensures proper externalization of configuration.

To comply with this principle, configuration should not be embedded within the code or tracked in the same codebase, except for default configuration, which can be bundled with the application. Other configurations can still be managed using separate files, but they should be stored in a distinct repository.

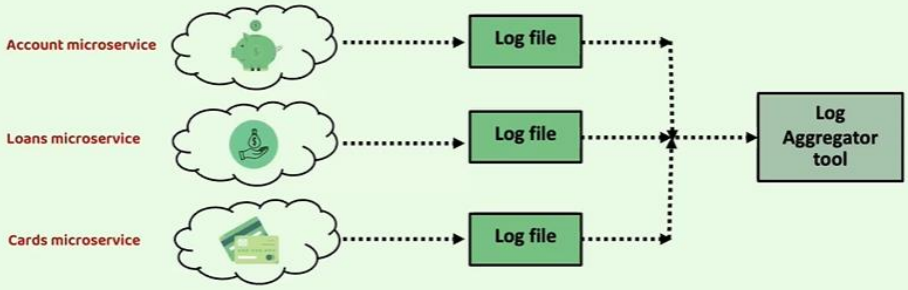
The methodology recommends utilizing environment variables to store configuration. This enables deploying the same application in different environments while adapting its behavior based on the specific environment's configuration.



15-Factor methodology - Logs

eazy bytes

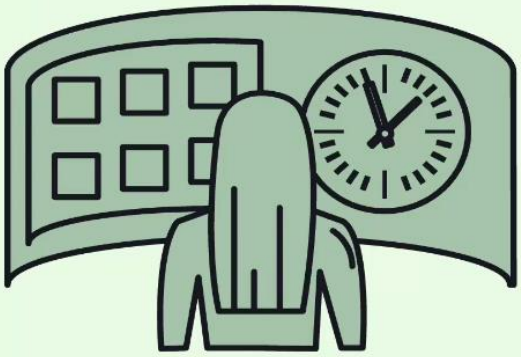
In a cloud-native application, log routing and storage are not the application's concern. Instead, applications should direct their logs to the standard output, treating them as sequentially ordered events based on time. The responsibility of log storage and rotation is now shifted to an external tool, known as a log aggregator. This tool retrieves, gathers, and provides access to the logs for inspection purposes.



15-Factor methodology - Disposability

eazy bytes

In a traditional environment, ensuring the continuous operation of applications is a top priority, striving to prevent any terminations. However, in a cloud environment, such meticulous attention is not necessary. Applications in the cloud are considered ephemeral, meaning that if a failure occurs and the application becomes unresponsive, it can be terminated and replaced with a new instance. Similarly, during high-load periods, additional instances of the application can be spun up to handle the increased workload. This concept is referred to as application disposability, where applications can be started or stopped as needed.



To effectively manage application instances in this dynamic environment, it is crucial to design them for quick startup when new instances are required and for graceful shutdown when they are no longer needed. A fast startup enables system elasticity, ensuring robustness and resilience. Without fast startup capabilities, performance and availability issues may arise.

A graceful shutdown involves the application, upon receiving a termination signal, ceasing to accept new requests, completing any ongoing ones, and then exiting. This process is straightforward for web processes. However, for worker processes or other types, it involves returning any pending jobs to the work queue before exiting.

Docker containers along with an orchestrator like Kubernetes inherently satisfy this requirement.

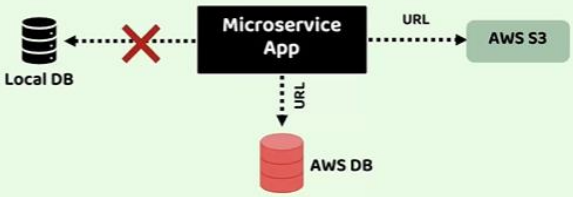
15-Factor methodology - Backing services

eazy bytes

Backing services refer to external resources that an application relies on to provide its functionality. These resources can include databases, message brokers, caching systems, SMTP servers, FTP servers, or RESTful web services. By treating these services as attached resources, you can modify or replace them without needing to make changes to the application code.

Consider the usage of databases throughout the software development life cycle. Typically, different databases are used in different stages such as development, testing, and production. By treating the database as an attached resource, you can easily switch to a different service depending on the environment. This attachment is achieved through resource binding, which involves providing necessary information like a URL, username, and password for connecting to the database.

In the below example, we can see that a local DB can be swapped easily to a third-party DB like AWS DB with out any code changes,



15-Factor methodology - Environment parity

eazy bytes

Environment parity aims to minimize differences between various environments & avoiding costly shortcuts. Here, the adoption of containers can greatly contribute by promoting the same execution environment.

There are three gaps that this factor addresses:

Time gap: The time it takes for a code change to be deployed can be significant. The methodology encourages automation and continuous deployment to reduce the time between code development and production deployment.

People gap: Developers create applications, while operators handle their deployment in production. To bridge this gap, a DevOps culture promotes collaboration between developers and operators, fostering the "you build it, you run it" philosophy.

Tools gap: Handling of backing services differs across environments. For instance, developers might use the H2 database locally but PostgreSQL in production. To achieve environment parity, it is recommended to use the same type and version of backing services across all environments.

15-Factor methodology - Administrative processes

eazy bytes

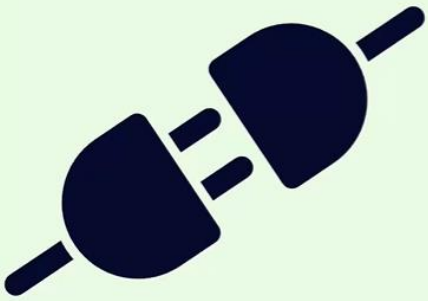
Management tasks required to support applications, such as database migrations, batch jobs, or maintenance tasks, should be treated as isolated processes. Similar to application processes, the code for these administrative tasks should be version controlled, packaged alongside the application, and executed within the same environment.

It is advisable to consider administrative tasks as independent microservices that are executed once and then discarded, or as functions configured within a stateless platform to respond to specific events. Alternatively, they can be integrated directly into the application, activated by calling a designated endpoint.

15-Factor methodology - Port binding

eazy bytes

Cloud native applications, adhering to the 15-Factor methodology, should be self-contained and expose their services through port binding. In production environments, routing services may be employed to translate requests from public endpoints to the internally port-bound services.

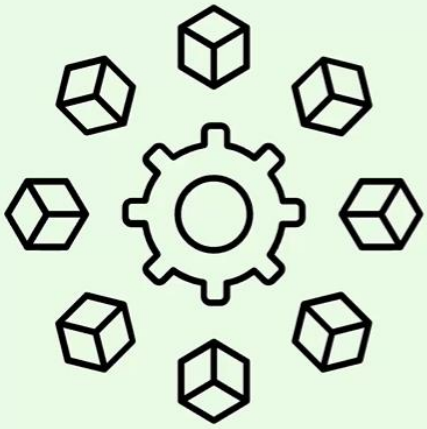


An application is considered self-contained when it doesn't rely on an external server within the execution environment. For instance, a Java web application might typically run within a server container like Tomcat, Jetty, or Undertow. In contrast, a cloud native application does not depend on the presence of a Tomcat server in the environment; it manages the server as a dependency within itself. For example, Spring Boot enables the usage of an embedded server, where the application incorporates the server instead of relying on its availability in the execution environment. Consequently, each application is mapped to its own server, diverging from the traditional approach of deploying multiple applications on a single server.

The services offered by the application are then exposed through port binding. For instance, a web application binds its HTTP services to a specific port and can potentially serve as a backing service for another application. This is a common practice within cloud native systems.

15-Factor methodology - Stateless processes

eazy bytes



Cloud native applications are often developed with high scalability in mind. One of the key principles to achieve scalability is designing applications as stateless processes and adopting a share-nothing architecture. This means that no state should be shared among different instances of the application. It is important to evaluate whether any data would be lost if an instance of the application is destroyed and recreated. If data loss would occur, then the application is not truly stateless.

However, it's important to note that some form of state management is necessary for applications to be functional. To address this, we design applications to be stateless and delegate the handling and storage of state to specific stateful services, such as data stores. In other words, a stateless application relies on a separate backing service to manage and store the required state, while the application itself remains stateless. This approach allows for better scalability and flexibility while ensuring that necessary state is still maintained and accessible when needed.

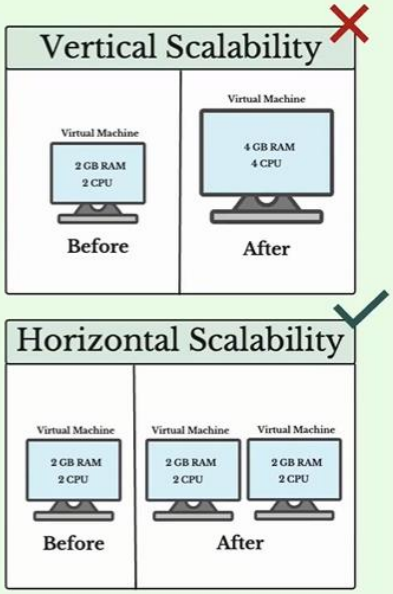
15-Factor methodology - Concurrency

eazy bytes

Scalability is not solely achieved by creating stateless applications. While statelessness is important, scalability also requires the ability to serve a larger number of users. This means that applications should support concurrent processing to handle multiple users simultaneously.

According to the 15-Factor methodology, processes play a crucial role in application design. These processes should be horizontally scalable, distributing the workload across multiple processes on different machines. This concurrency is only feasible when applications are stateless. In Java Virtual Machine (JVM) applications, concurrency is typically managed through the use of multiple threads, which are available from thread pools.

Processes can be categorized based on their respective types. For instance, there are web processes responsible for handling HTTP requests, as well as worker processes that execute scheduled background jobs. By classifying processes and optimizing their concurrency, applications can effectively scale and handle increased workloads.



15-Factor methodology - Telemetry

eazy bytes

Observability is a fundamental characteristic of cloud native applications. With the inherent complexity of managing a distributed system in the cloud, it becomes essential to have access to accurate and comprehensive data from each component of the system. This data enables remote monitoring of the system's behavior and facilitates effective management of its intricacies. Telemetry data, such as logs, metrics, traces, health status, and events, plays a vital role in providing this visibility.

In Kevin Hoffman's analogy, he emphasizes the significance of telemetry by comparing applications to space probes. Just like telemetry is crucial for monitoring and controlling space probes remotely, the same concept applies to applications. To effectively monitor and control applications remotely, you need various types of telemetry data.

Consider the kind of telemetry that would be necessary to ensure remote monitoring and control of your applications. This includes information such as detailed logs for troubleshooting, metrics to measure performance, traces to understand request flows, health status to assess system well-being, and events to capture significant occurrences. By gathering and utilizing these types of telemetry data, you can gain valuable insights into your applications and make informed decisions to manage them effectively from a remote location.

15-Factor methodology - Authentication and authorization

eazy bytes

Security is a critical aspect of a software system, yet it often doesn't receive the necessary emphasis it deserves. To uphold a zero-trust approach, it is essential to ensure the security of every interaction within the system, encompassing architectural and infrastructural levels. While security involves more than just authentication and authorization, these aspects serve as a solid starting point.

Authentication enables us to track and verify the identity of users accessing the application. By authenticating users, we can then proceed to evaluate their permissions and determine if they have the necessary authorization to perform specific actions. Implementing identity and access management standards can greatly enhance security. Notable examples include OAuth 2.1 and OpenID Connect, which we will explore in this course.

CONFIGURATION MANAGEMENT IN MICROSERVICES

eazy bytes

SEPARATION OF CONFIGs/PROPERTIES

How do we separate the configurations/properties from the microservices so that same Docker image can be deployed in multiple envs.

INJECT CONFIGs/PROPERTIES

How do we inject configurations/properties that microservice needed during start up of the service

MAINTAIN CONFIGs/PROPERTIES

How do we maintain configurations/properties in a centralized repository along with versioning of them

There are multiple solutions available in Spring Boot ecosystem to handle this challenge. Below are the solutions. Let's try to identify which one suites for microservices

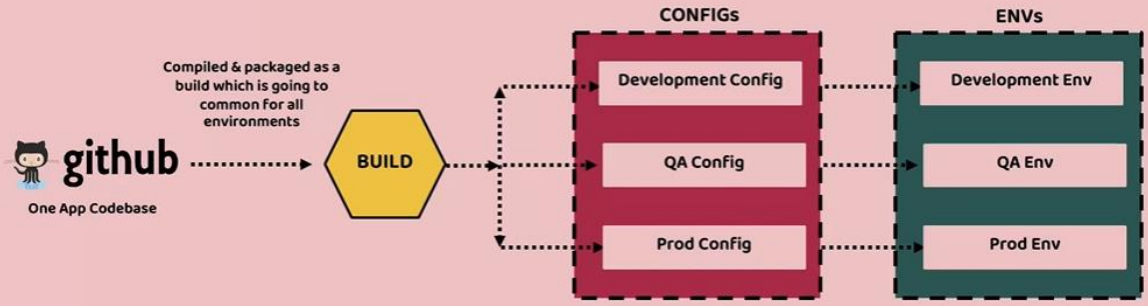
1) Configuring Spring Boot with properties and profiles
2) Applying external configuration with Spring Boot
3) Implementing a configuration server with Spring Cloud Config Server

HOW CONFIGURATIONs HANDLED IN TRADITIONAL APPs & MICROSERVICES

eazy bytes

Traditional applications were typically bundled together with their source code and various configuration files that contained environment-specific data. This meant that updating the configuration required rebuilding the entire application, or creating separate builds for each environment. As a result, there was no guarantee that the application would behave consistently across different environments, leading to potential issues when moving from staging to production.

According to the 15-Factor methodology, configuration encompasses any element likely to change between deployments, such as credentials, resource handles, and service URLs. Cloud native applications address this challenge by maintaining the immutability of the application artifact across environments. Regardless of the deployment environment, the application build remains unchanged. In cloud native applications, each deployment involves combining the build with specific configuration data. This allows the same build to be deployed to multiple environments while accommodating different configuration requirements, as shown below,



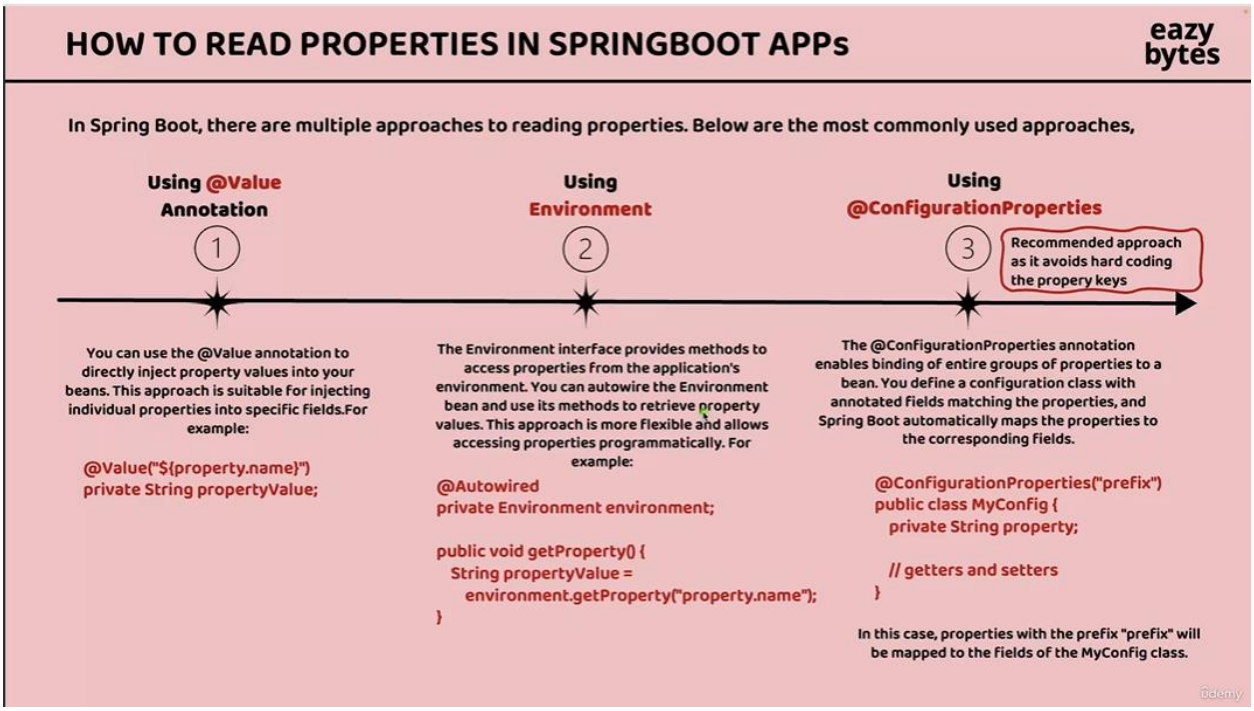
eazy bytes

HOW CONFIGURATIONS WORK IN SPRINGBOOT ?

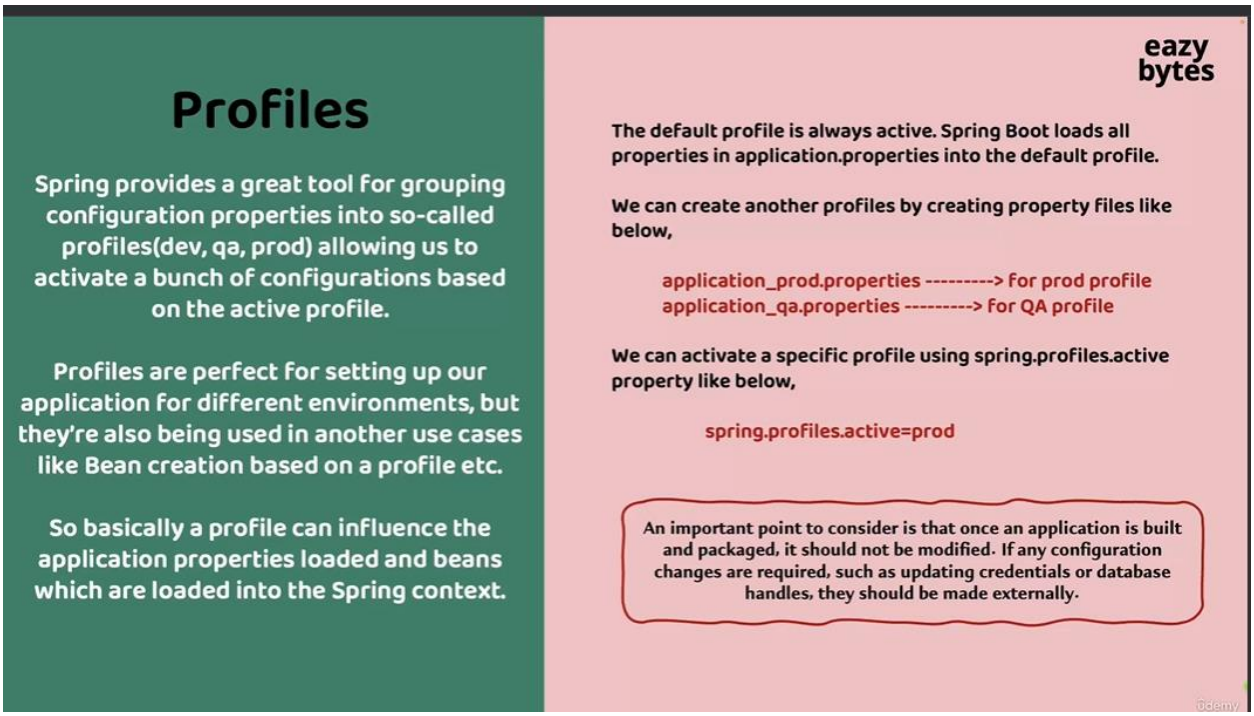
Spring Boot lets you externalize your configuration so that you can work with the same application code in different environments. You can use a variety of external configuration sources, include Java properties files, YAML files, environment variables, and command-line arguments.

- ✓ By default, Spring Boot look for the configurations or properties inside application.properties/yaml present in the classpath location. But we can have other property files as well and make SpringBoot to read from them.
- ✓ Spring Boot uses a very particular order that is designed to allow sensible overriding of values. Properties are considered in the following order (with values from lower items overriding earlier ones):

- Properties present inside files like application.properties
- OS Environmental variables
- Java System properties (System.getProperties())
- JNDI attributes from java:comp/env
- ServletContext init parameters
- ServletConfig init parameters
- Command line arguments



Code Available in section 6



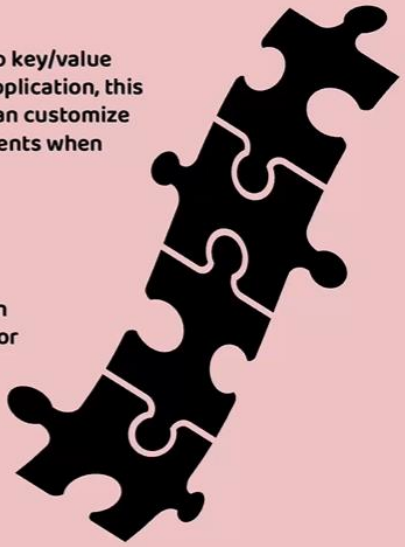
How to externalize configurations using command-line arguments ?

eazy bytes

Spring Boot automatically converts command-line arguments into key/value pairs and adds them to the Environment object. In a production application, this becomes the property source with the highest precedence. You can customize the application configuration by specifying command-line arguments when running the JAR you built earlier.

```
java -jar accounts-service-0.0.1-SNAPSHOT.jar --build.version="1.1"
```

The command-line argument follows the same naming convention as the corresponding Spring property, with the familiar -- prefix for CLI arguments.



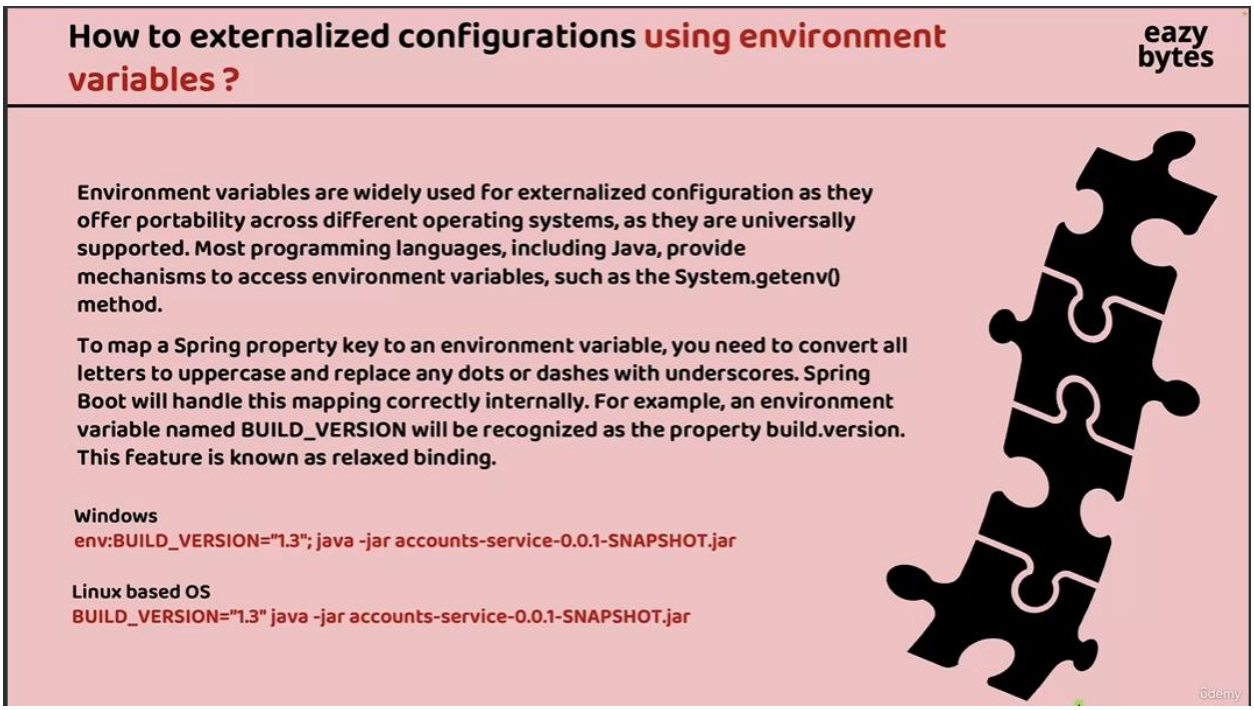
How to externalized configurations using JVM system properties ?

eazy bytes

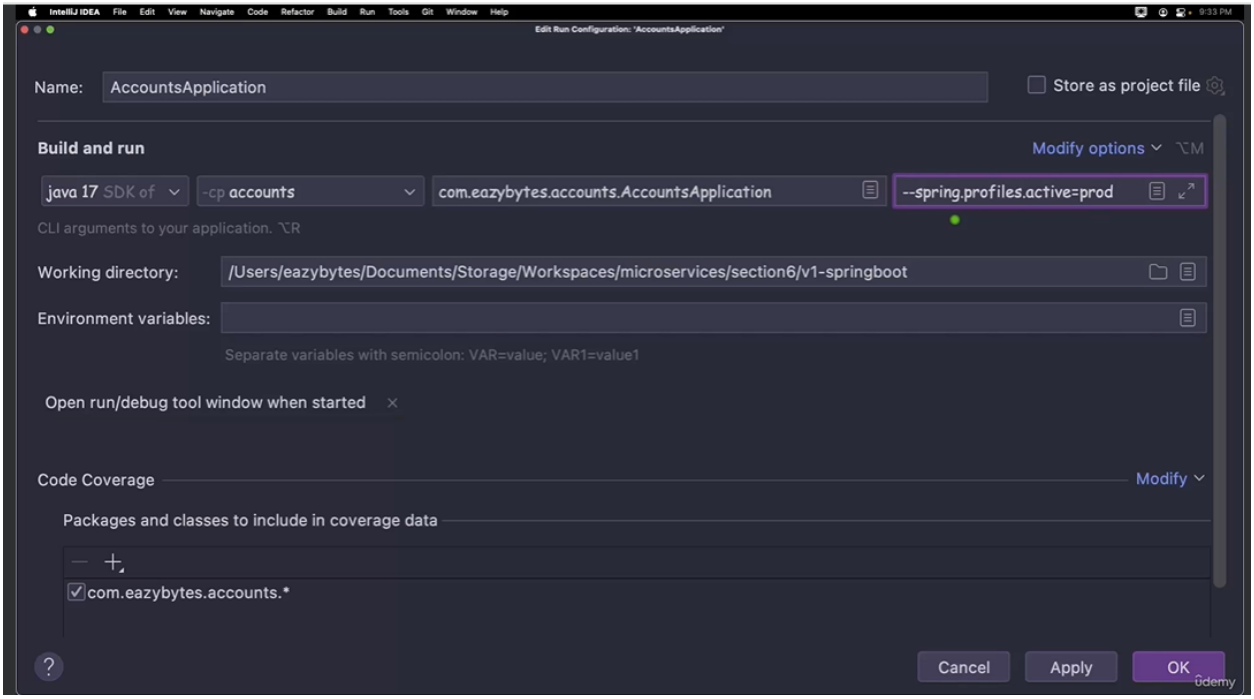
JVM system properties, similar to command-line arguments, can override Spring properties with a lower priority. This approach allows for externalizing the configuration without the need to rebuild the JAR artifact. The JVM system property follows the same naming convention as the corresponding Spring property, prefixed with -D for JVM arguments. In the application, the message defined as a JVM system property will be utilized, taking precedence over property files.

```
java -Dbuild.version="1.2" -jar accounts-service-0.0.1-SNAPSHOT.jar
```

In the scenario where both a JVM system property and a command-line argument are specified, the precedence rules dictate that Spring will prioritize the value provided as a command-line argument. This means that the value specified through the CLI will be utilized by the application, taking precedence over the JVM properties.

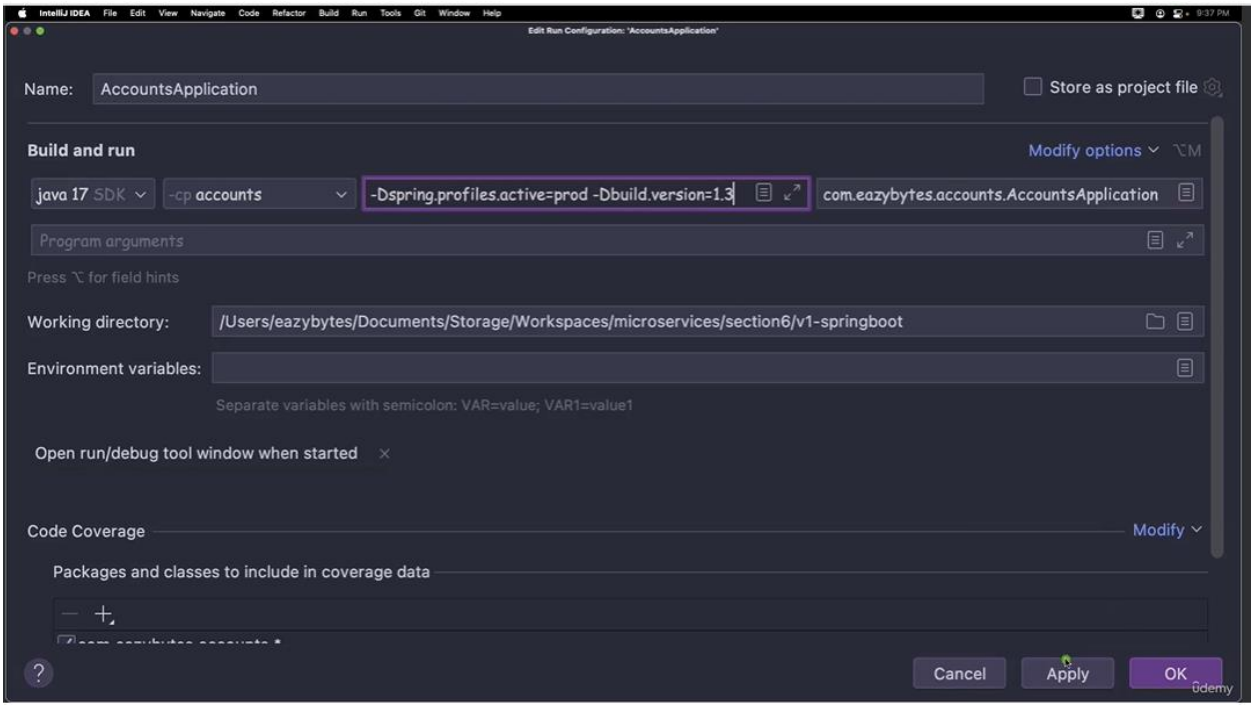


How to activate a profile using command line argument



How to pass JVM variable at runtime

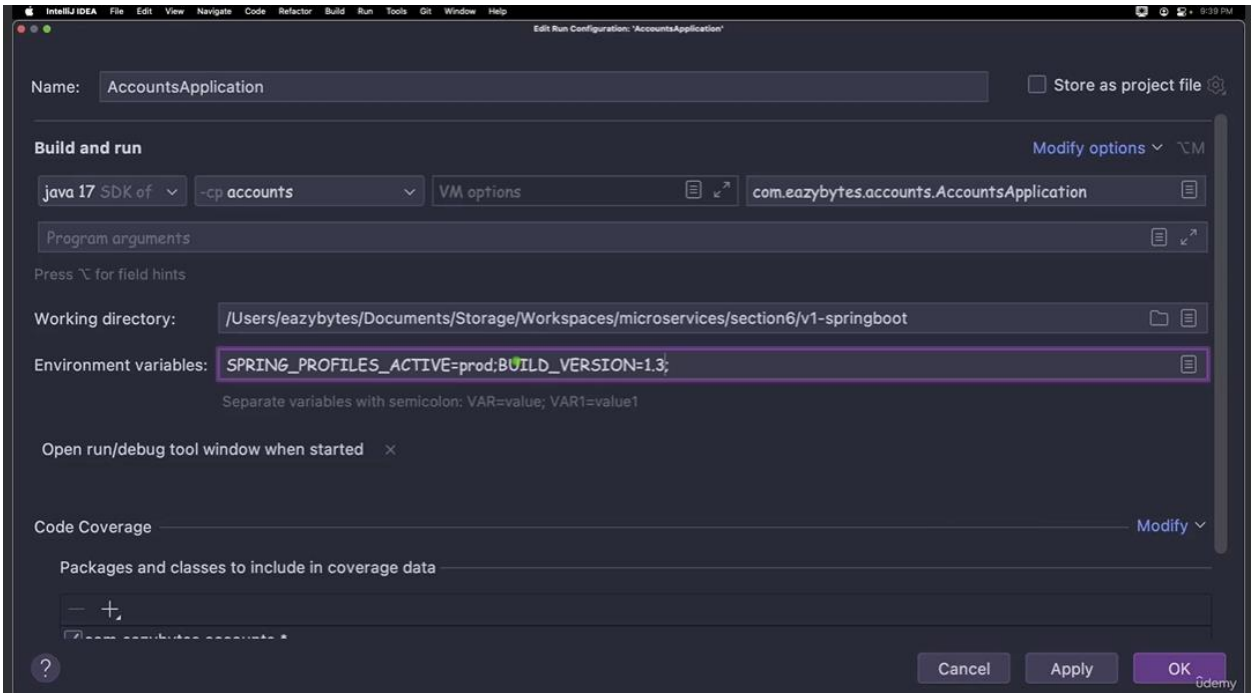
-Dspring.profile.active=prod -Dbuild.version=4.1



How to pass configuration using Environment Variable

Replace dot with underscore and make all in upper case and give semicolon (;) for each variable

SPRING_PROFILE_ACTIVE=prod; BUILD_VERSION=3.1



Drawbacks of externalized configurations using SpringBoot alone

eazy bytes

1. CLI arguments, JVM properties, and environment variables are effective ways to externalize configuration and maintain the immutability of the application build. However, using these approaches often involves executing separate commands and manually setting up the application, which can introduce potential errors during deployment.
2. Given that configuration data evolves and requires changes, similar to application code, what strategies should be employed to store, track revisions and audit the configuration used in a release?
3. In scenarios where environment variables lack granular access control features, how can you effectively control access to configuration data?

4. When the number of application instances grows, handling configuration in a distributed manner for each instance becomes challenging. How can such challenges be overcome?
5. Considering that neither Spring Boot properties nor environment variables support configuration encryption, how should secrets be managed securely?
6. After modifying configuration data, how can you ensure that the application can read it at runtime without necessitating a complete restart?

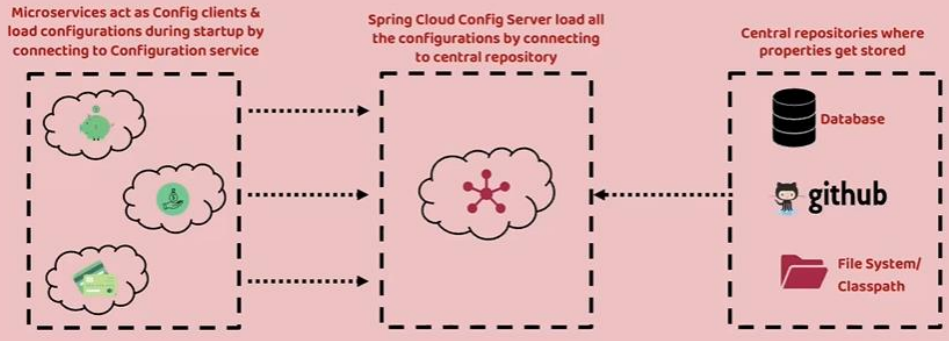
Spring Cloud Config

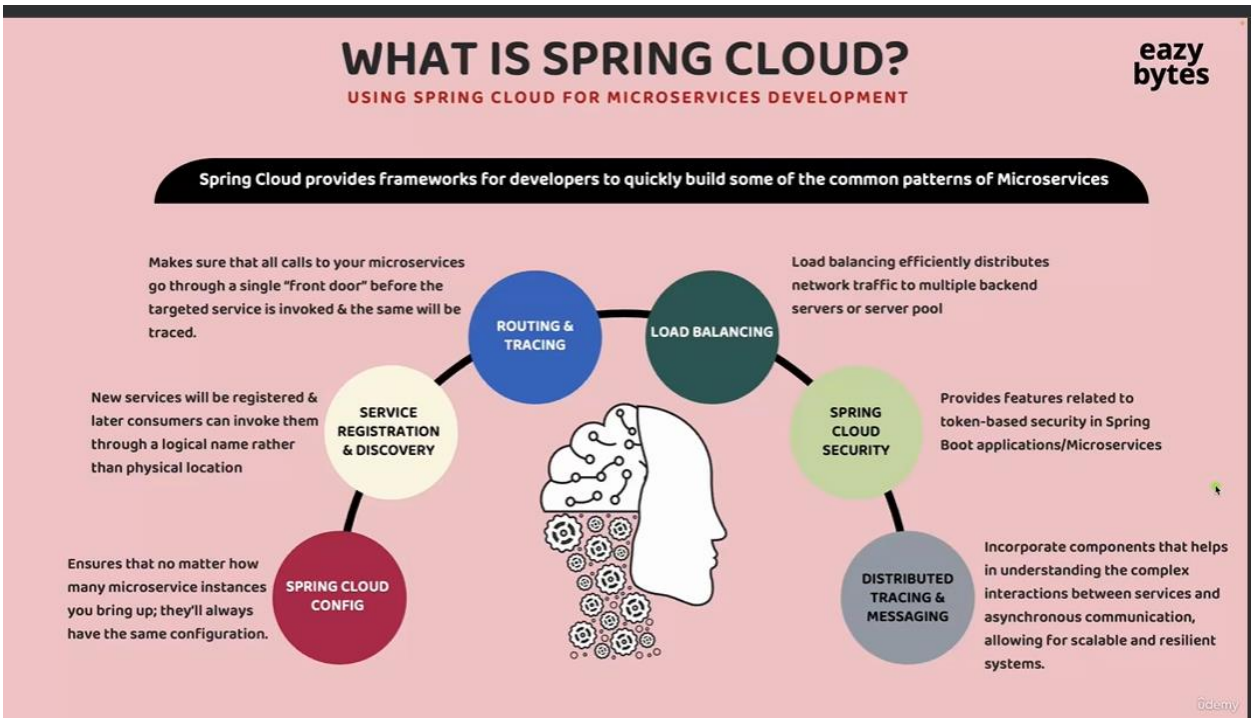
eazy bytes

A centralized configuration server with Spring Cloud Config can overcome all the drawbacks that we discussed in the previous slide. Spring Cloud Config provides server and client-side support for externalized configuration in a distributed system. With the Config Server you have a central place to manage external properties for applications across all environments.

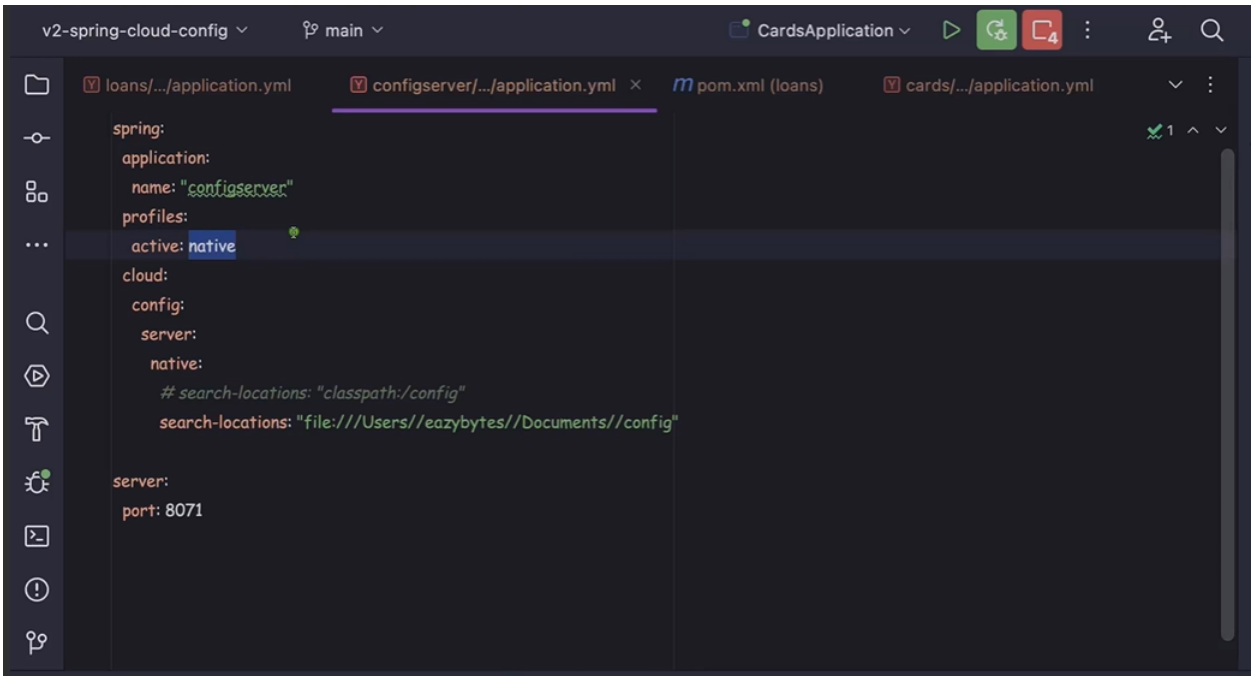
Centralized configuration revolves around two core elements:

- A data store designed to handle configuration data, ensuring durability, version management, and potentially access control.
- A server that oversees the configuration data within the data store, facilitating its management and distribution to multiple applications.

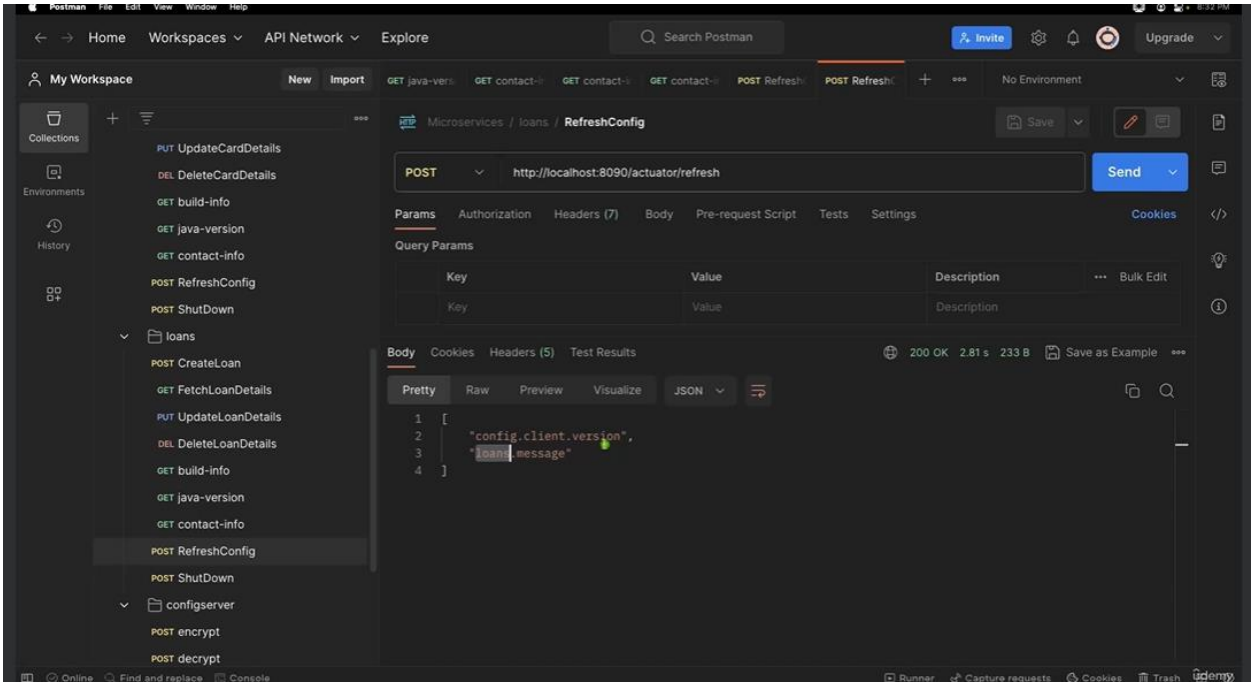




Config server configuration for local set up of configuration file of different microservices



To reflect the changes at run time we need to invoke refresh api for all microservices



Refresh configurations at runtime using /refresh path

eazy bytes

What occurs when new updates are committed to the Git repository supporting the Config Service? In a typical Spring Boot application, modifying a property would require a restart. However, Spring Cloud Config introduces the capability to dynamically refresh the configuration in client applications during runtime. When a change is pushed to the configuration repository, all integrated applications connected to the config server can be notified, prompting them to reload the relevant portions affected by the configuration modification.

Let's see an approach for refreshing the configuration, which involves sending a specific POST request to a running instance of the microservice. This request will initiate the reloading of the modified configuration data, enabling a hot reload of the application. Below are the steps to follow,

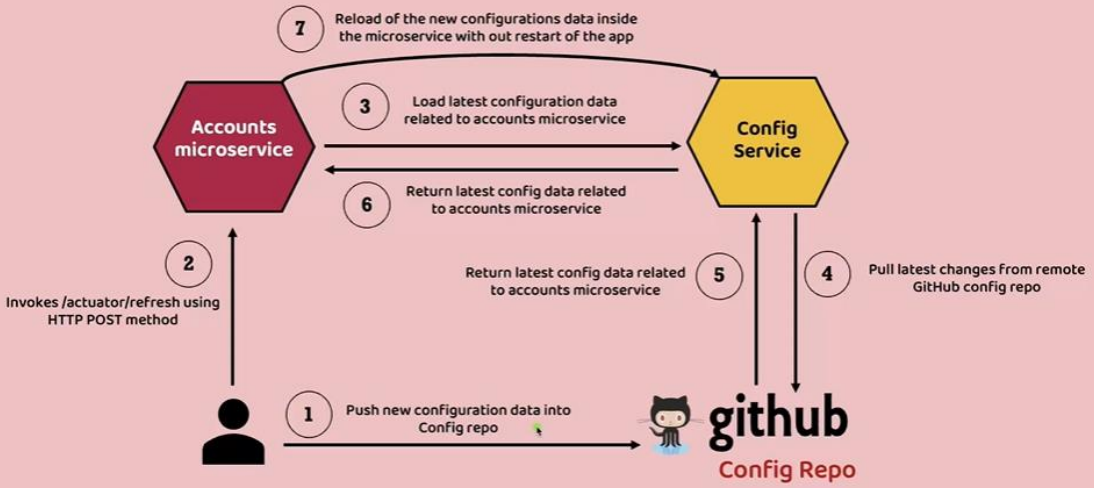
1. **Add actuator dependency in the Config Client services:** Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards to expose the /refresh endpoint

2. **Enable /refresh API :** The Spring Boot Actuator library provides a configuration endpoint called "/actuator/refresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

```
management:
  endpoints:
    web:
      exposure:
        include: refresh
```

Refresh configurations at runtime using /refresh path

eazy bytes



You invoked the refresh mechanism on Accounts Service, and it worked fine, since it was just one application with 1 instance. How about in production where there may be multiple services ?If a project has many microservices, then team may prefer to have an automated and efficient method for refreshing configuration instead of manually triggering each application instance. Let's evaluate other options that we have

Refresh configurations at runtime using Spring Cloud Bus

eazy bytes

Spring Cloud Bus, available at https://spring.io/projects/spring-cloud-bus, facilitates seamless communication between all connected application instances by establishing a convenient event broadcasting channel. It offers an implementation for AMQP brokers, such as RabbitMQ, and Kafka, enabling efficient communication across the application ecosystem.

Below are the steps to follow,

1. Add actuator dependency in the Config Server & Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans and cards to expose the /busrefresh endpoint

2. Enable / busrefresh API : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/busrefresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

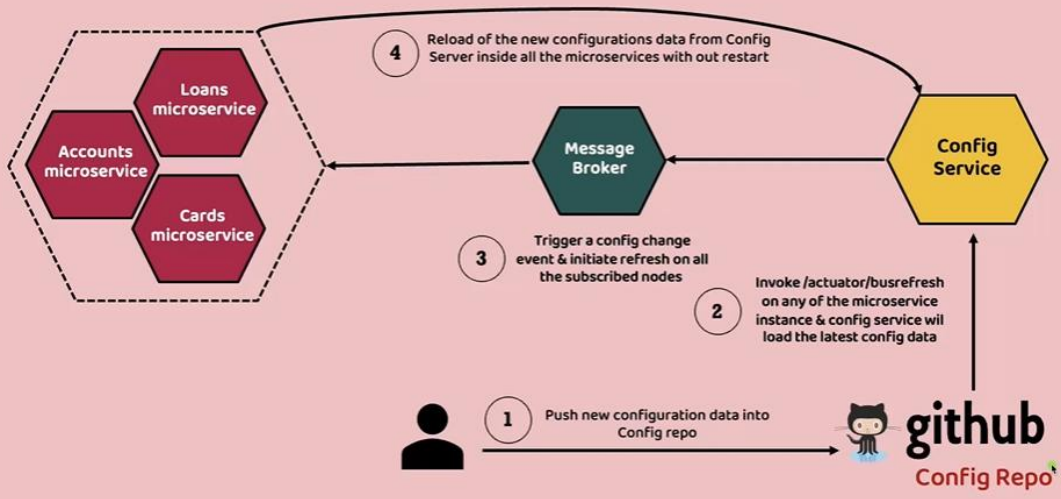
```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

3. Add Spring Cloud Bus dependency in the Config Server & Client services: Add Spring Cloud Bus dependency (spring-cloud-starter-bus-amqp) inside pom.xml of the individual microservices like accounts, loans, cards and Config server

4. Set up a RabbitMQ: Using Docker, setup RabbitMQ service. If the service is not started with default values, then configure the rabbitmq connection details in the application.yml file of all the individual microservices and Config server

Refresh configurations at runtime using Spring Cloud Bus

eazy bytes



Though this approach reduce manual work to a great extent, but still there is a single manual step involved which is invoking the /actuator/busrefresh on any of the microservice instance. Let's see how we can avoid and completely automate the process.

Refresh configurations at runtime using Spring Cloud Bus & Spring Cloud Config Monitor

eazy bytes

Spring Cloud Config offers the Monitor library, which enables the triggering of configuration change events in the Config Service. By exposing the /monitor endpoint, it facilitates the propagation of these events to all listening applications via the Bus. The Monitor library allows push notifications from popular code repository providers such as GitHub, GitLab, and Bitbucket. You can configure webhooks in these services to automatically send a POST request to the Config Service after each new push to the configuration repository. Below are the steps to follow,

1. Add actuator dependency in the Config Server & Client services: Add Spring Boot Actuator dependency inside pom.xml of the individual microservices like accounts, loans, cards and Config server to expose the /busrefresh endpoint

2. Enable / busrefresh API : The Spring Boot Actuator library provides a configuration endpoint called "/actuator/busrefresh" that can trigger a refresh event. By default, this endpoint is not exposed, so you need to explicitly enable it in the application.yml file using the below config,

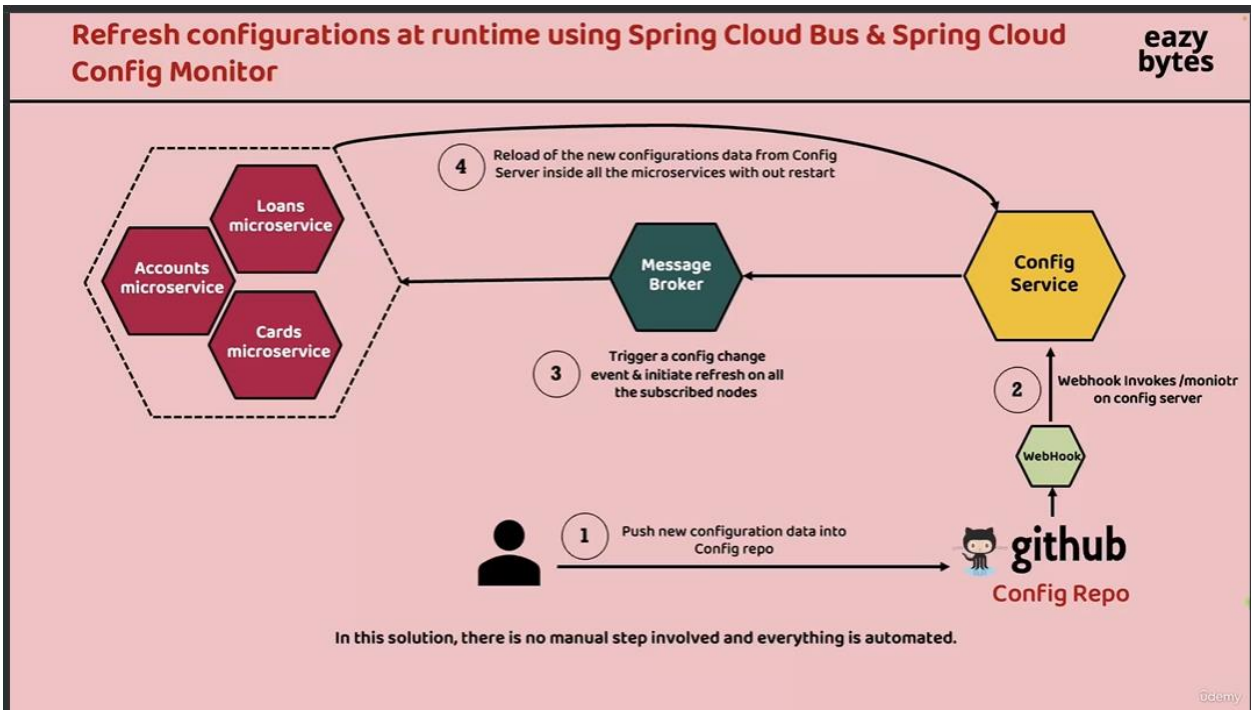
```
management:
  endpoints:
    web:
      exposure:
        include: busrefresh
```

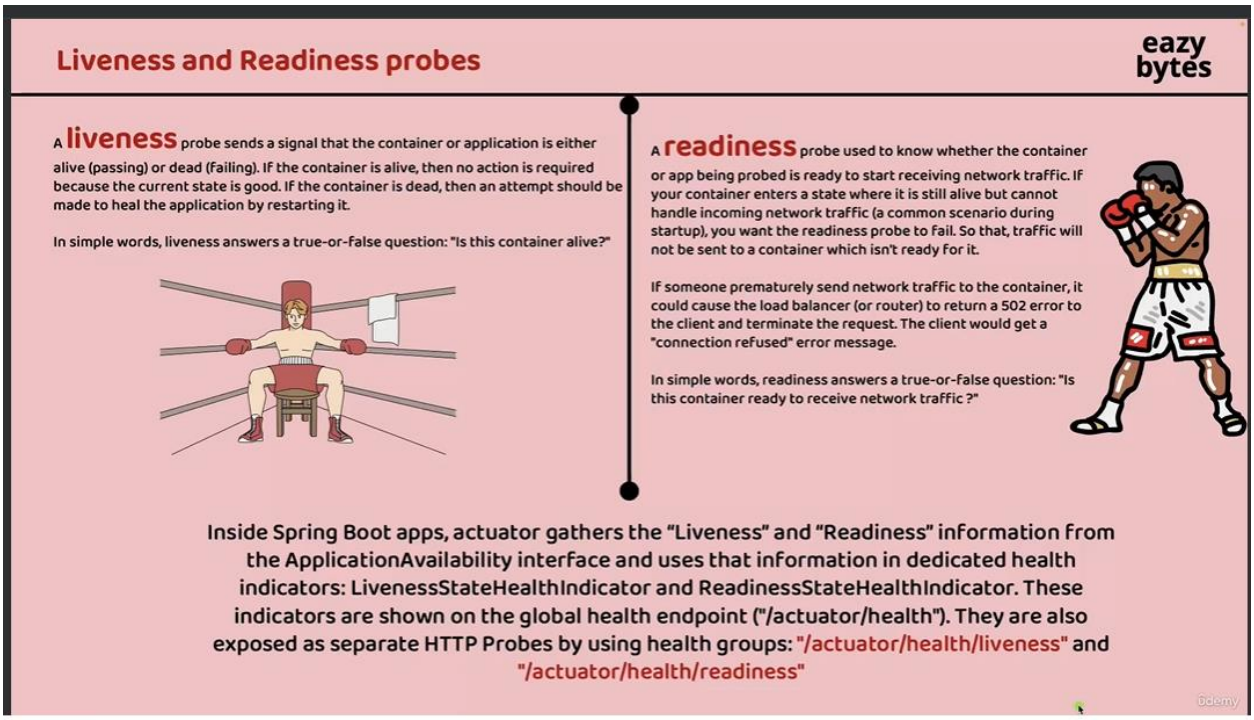
3. Add Spring Cloud Bus dependency in the Config Server & Client services: Add Spring Cloud Bus dependency (spring-cloud-starter-bus-amqp) inside pom.xml of the individual microservices like accounts, loans, cards and Config server

4. Add Spring Cloud Config monitor dependency in the Config Server : Add Spring Cloud Config monitor dependency (spring-cloud-config-monitor) inside pom.xml of Config server and this exposes /monitor endpoint

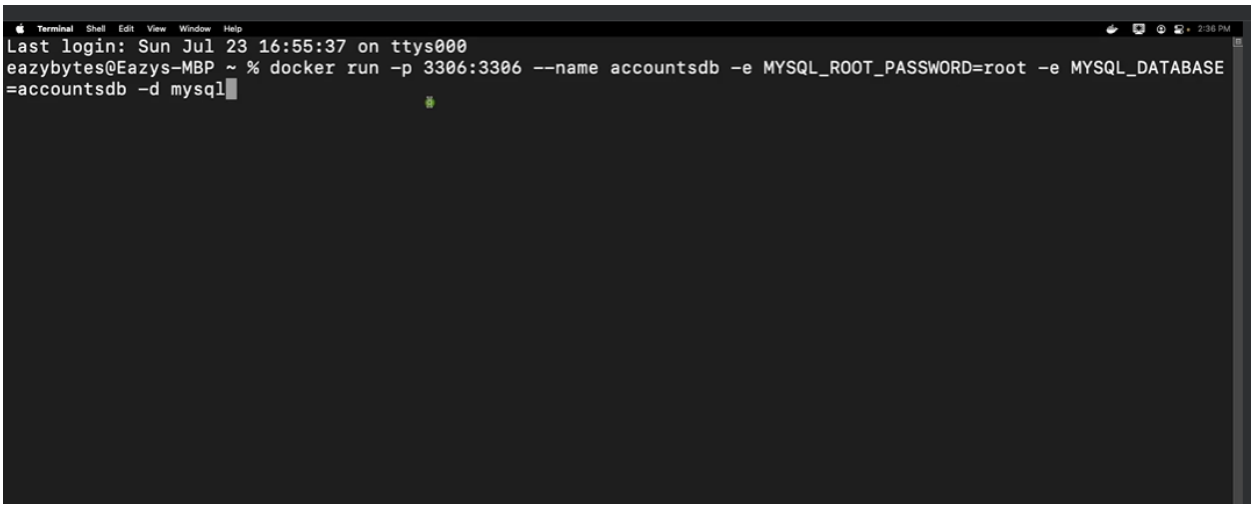
5. Set up a RabbitMQ: Using Docker, setup RabbitMQ service. If the service is not started with default values, then configure the rabbitmq connection details in the application.yml file of all the individual microservices and Config server

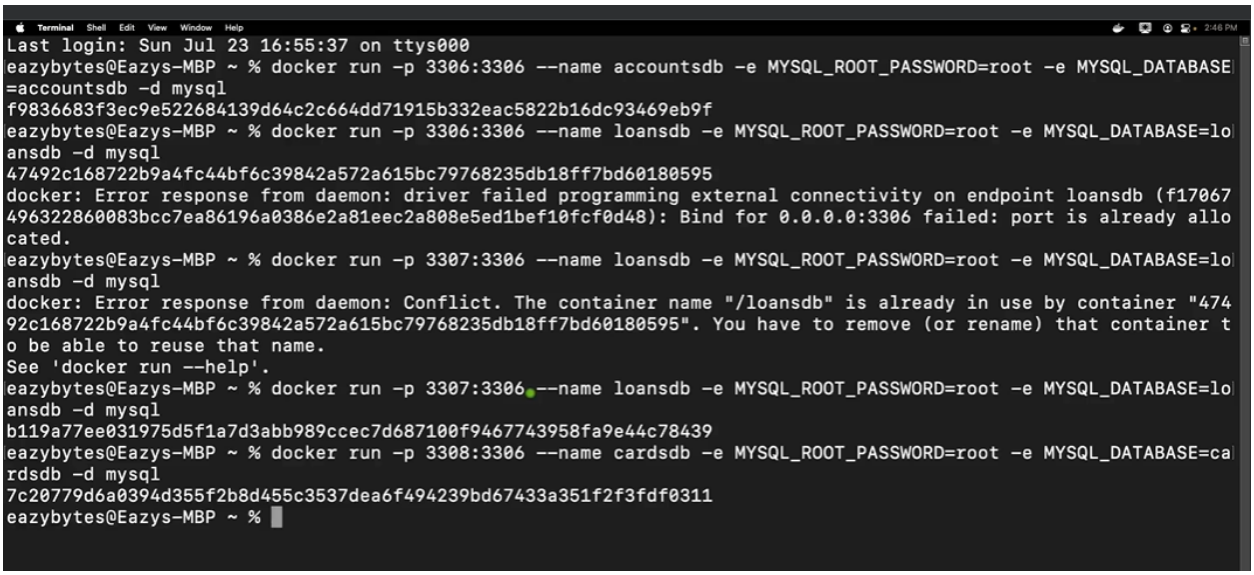
6. Set up a WebHook in GitHub: Set up a webhook to automatically send a POST request to Config Service /monitor path after each new push to the config repo.

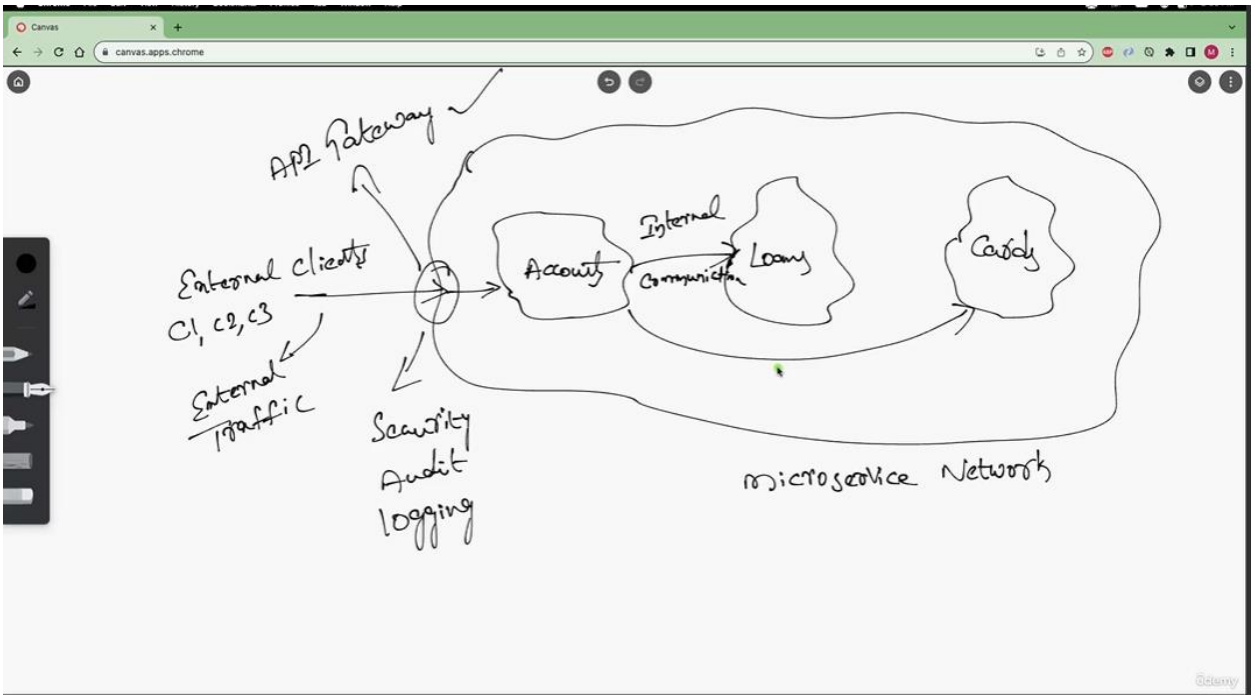


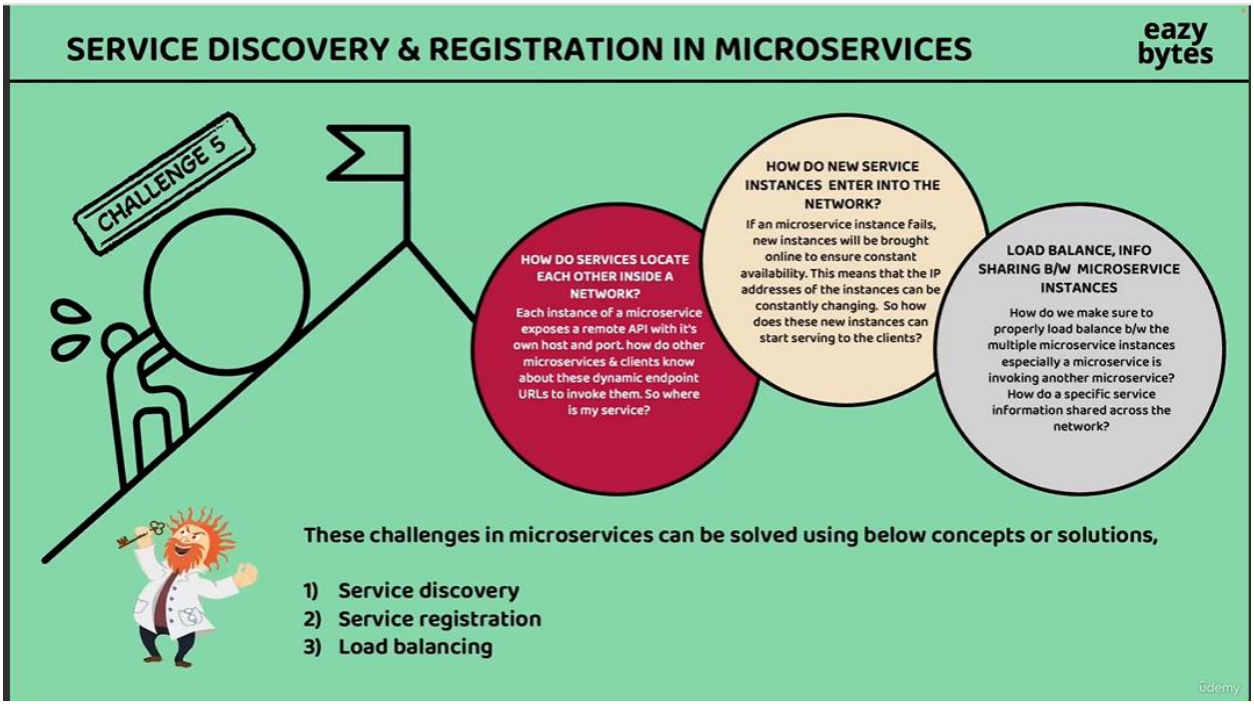


Docker command to create My_Sql db inside docker


Terminal Shell Edit View Window Help
2:36 PM
Last login: Sun Jul 23 16:55:37 on ttys000
eazybytes@Eazys-MBP ~ % docker run -p 3306:3306 --name accountsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE
=accountsdb -d mysql


Terminal Shell Edit View Window Help
2:46 PM
Last login: Sun Jul 23 16:55:37 on ttys000
eazybytes@Eazys-MBP ~ % docker run -p 3306:3306 --name accountsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE
=accountsdb -d mysql
f9836683f3ec9e522684139d64c2c664dd71915b332eac5822b16dc93469eb9f
eazybytes@Eazys-MBP ~ % docker run -p 3306:3306 --name loansdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=lo
ansdb -d mysql
47492c168722b9a4fc44bf6c39842a572a615bc79768235db18ff7bd60180595
docker: Error response from daemon: driver failed programming external connectivity on endpoint loansdb (f17067
496322860083bcc7ea86196a0386e2a81eec2a808e5ed1bef10fcf0d48): Bind for 0.0.0.0:3306 failed: port is already allo
cated.
eazybytes@Eazys-MBP ~ % docker run -p 3307:3306 --name loansdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=lo
ansdb -d mysql
docker: Error response from daemon: Conflict. The container name "/loansdb" is already in use by container "474
92c168722b9a4fc44bf6c39842a572a615bc79768235db18ff7bd60180595". You have to remove (or rename) that container t
o be able to reuse that name.
See 'docker run --help'.
eazybytes@Eazys-MBP ~ % docker run -p 3307:3306 --name loansdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=lo
ansdb -d mysql
b119a77ee031975d5f1a7d3abb989ccec7d687100f9467743958fa9e44c78439
eazybytes@Eazys-MBP ~ % docker run -p 3308:3306 --name cardsdb -e MYSQL_ROOT_PASSWORD=root -e MYSQL_DATABASE=ca
rdsdb -d mysql
7c20779d6a0394d355f2b8d455c3537dea6f494239bd67433a351f2f3fdf0311
eazybytes@Eazys-MBP ~ %

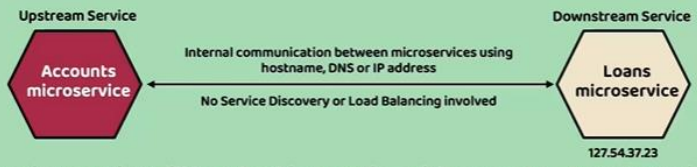

Canvas
canvas.apps.chrome
API Gateway
External clients
C1, C2, C3
External traffic
Security
Audit
Logging
Accounts
Internal
communication
Loans
Cards
microservice Network


SERVICE DISCOVERY & REGISTRATION IN MICROSERVICES
eazy bytes
CHALLENGE 5
HOW DO SERVICES LOCATE EACH OTHER INSIDE A NETWORK?
Each instance of a microservice exposes a remote API with it's own host and port. how do other microservices & clients know about these dynamic endpoint URLs to invoke them. So where is my service?
HOW DO NEW SERVICE INSTANCES ENTER INTO THE NETWORK?
If an microservice instance fails, new instances will be brought online to ensure constant availability. This means that the IP addresses of the instances can be constantly changing. So how does these new instances can start serving to the clients?
LOAD BALANCE, INFO SHARING B/W MICROSERVICE INSTANCES
How do we make sure to properly load balance b/w the multiple microservice instances especially a microservice is invoking another microservice? How do a specific service information shared across the network?
These challenges in microservices can be solved using below concepts or solutions,
1) Service discovery
2) Service registration
3) Load balancing

How service communication happens in Traditional apps ?

eazy bytes

Inside web network, When a service/app want to communicate with another service/app, it must be given the necessary information to locate it, such as an IP address or a DNS name. Let's examine the scenario of two services, Accounts and Loans. If there was only a single instance of Loans microservice, below figure illustrates how the communication between the two applications would occur.



Loans microservice will be a backing service with respect to Accounts microservice

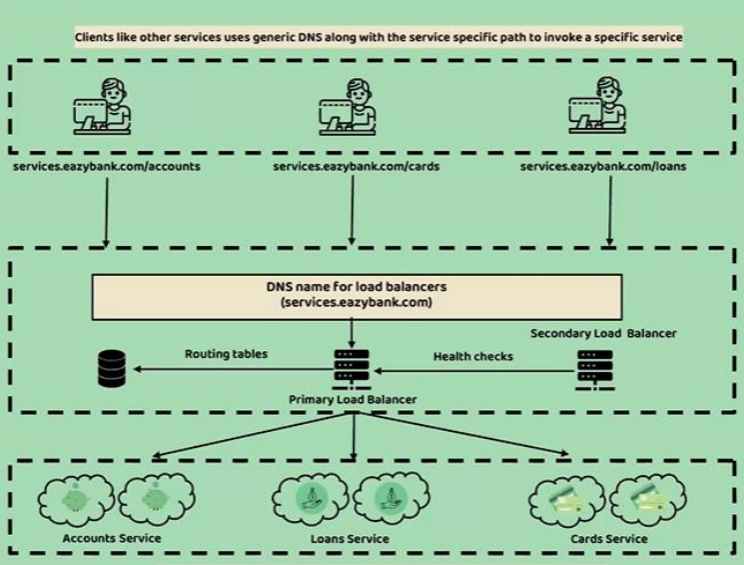
When there is only one instance of the Loans microservice running, managing the DNS name and its corresponding IP address mapping is straightforward. However, in a cloud environment, it is common to deploy multiple instances of a service, with each instance having its own unique IP address.

To address this challenge, one approach is to update DNS records with multiple IP addresses and rely on round-robin name resolution. This method directs requests to one of the IP addresses assigned to the service replicas in a rotating manner. However, this approach may not be suitable for microservices, as containers or services change frequently. This rapid change makes it difficult to maintain accurate DNS records and ensure efficient communication between microservices.

Unlike physical machines or long-running virtual machines, cloud-based service instances have shorter lifespans. These instances are designed to be disposable and can be terminated or replaced for various reasons, such as unresponsiveness. Furthermore, auto-scaling capabilities can be enabled to automatically adjust the number of application instances based on the workload.

How Traditional LoadBalancers works ?

eazy bytes



Traditional Service location resolution architecture using DNS & a load balancer

Limitations with Traditional LoadBalancers ?

eazy bytes

With traditional approach each instance of a service used to be deployed in one or more application servers. The number of these application servers was often static and even in the case of restoration it would be restored to the same state with the same IP and other configurations.

While this type of model works well with monolithic and SOA based applications with a relatively small number of services running on a group of static servers, it doesn't work well for cloud-based microservice applications for the following reasons,

- Limited horizontal scalability & licenses costs
- Single point of failure & Centralized chokepoints
- Manually managed to update any IPs, configurations
- Complex in nature & not containers friendly

The biggest challenge with traditional load balancers is that some one has to manually maintain the routing tables which is an impossible task inside the microservices network. Because containers/services are ephemeral in nature

How to solve the problem for cloud native applications ?

eazy bytes

For cloud native applications, service discovery is the perfect solution. It involves tracking and storing information about all running service instances in a service registry.

Whenever a new instance is created, it should be registered in the registry, and when it is terminated, it should be appropriately removed automatically.

The registry acknowledges that multiple instances of the same application can be active simultaneously. When an application needs to communicate with a backing service, it performs a lookup in the registry to determine the IP address to connect to. If multiple instances are available, a load-balancing strategy is employed to evenly distribute the workload among them.

Client-side service discovery and server-side service discovery are distinct approaches that address the service discovery problem in different contexts

How to solve the problem for cloud native applications ?

eazy bytes

In a modern microservice architecture, knowing the right network location of an application is a much more complex problem for the clients as service instances might have dynamically assigned IP addresses. Moreover the number instances may vary due to autoscaling and failures.

Microservices service discovery & registration is a way for applications and microservices to locate each other on a network. This includes,

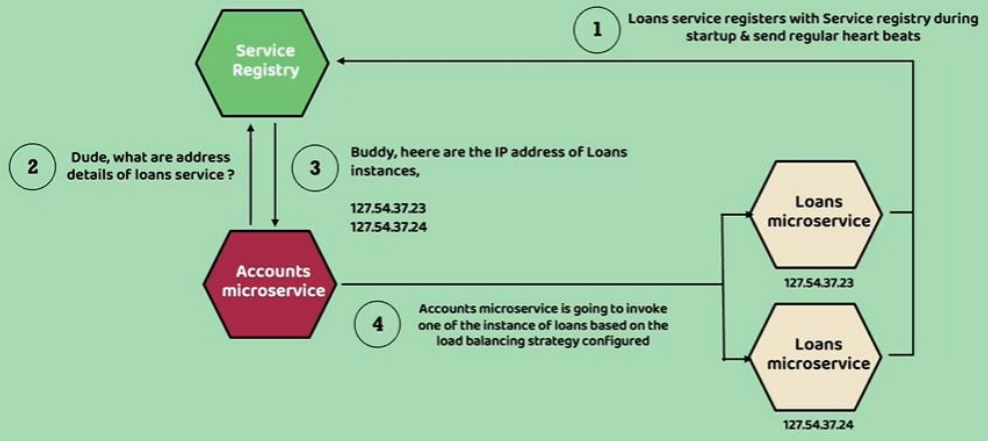
- A central server (or servers) that maintain a global view of addresses
- Microservices/clients that connect to the central server to register their address when they start & ready
- Microservices/clients need to send their heartbeats at regular intervals to central server about their health
- Microservices/clients that connect to the central server to deregister their address when they are about to shutdown

Service discovery & registrations deals with the problems about how microservices talk to each other, i.e. perform API calls.

Client-side service discovery and load balancing

eazy bytes

In client-side service discovery, applications are responsible for registering themselves with a service registry during startup and unregistering when shutting down. When an application needs to communicate with a backing service, it queries the service registry for the associated IP address. If multiple instances of the service are available, the registry returns a list of IP addresses. The client application then selects one based on its own defined load-balancing strategy. Below figure illustrates the workflow of this process



Client-side service discovery and load balancing

eazy bytes

Client-side service discovery is an architectural pattern where client applications are responsible for locating and connecting to services they depend on. In this approach, the client application communicates directly with a service registry to discover available service instances and obtain the necessary information to establish connections.

Here are the key aspects of client-side service discovery:

- **Service Registration:** Client applications register themselves with the service registry upon startup. They provide essential information about their location, such as IP address, port, and metadata, which helps identify and categorize the service.
- **Service Discovery:** When a client application needs to communicate with a specific service, it queries the service registry for available instances of that service. The registry responds with the necessary information, such as IP addresses and connection details.
- **Load Balancing:** Client-side service discovery often involves load balancing to distribute the workload across multiple service instances. The client application can implement a load-balancing strategy to select a specific instance based on factors like round-robin, weighted distribution, or latency.

The major advantage of client-side service discovery is load balancing can be implemented using various algorithms, such as round-robin, weighted round-robin, least connections, or even custom algorithms. A drawback is that client service discovery assigns more responsibility to developers. Also, it results in one more service to deploy and maintain (the service registry). Server-side discovery solutions solve these issues. We are going to discuss the same when we are talking about Kubernetes

Client-side service discovery and load balancing

eazy bytes

Client-side service discovery is an architectural pattern where client applications are responsible for locating and connecting to services they depend on. In this approach, the client application communicates directly with a service registry to discover available service instances and obtain the necessary information to establish connections.

Here are the key aspects of client-side service discovery:

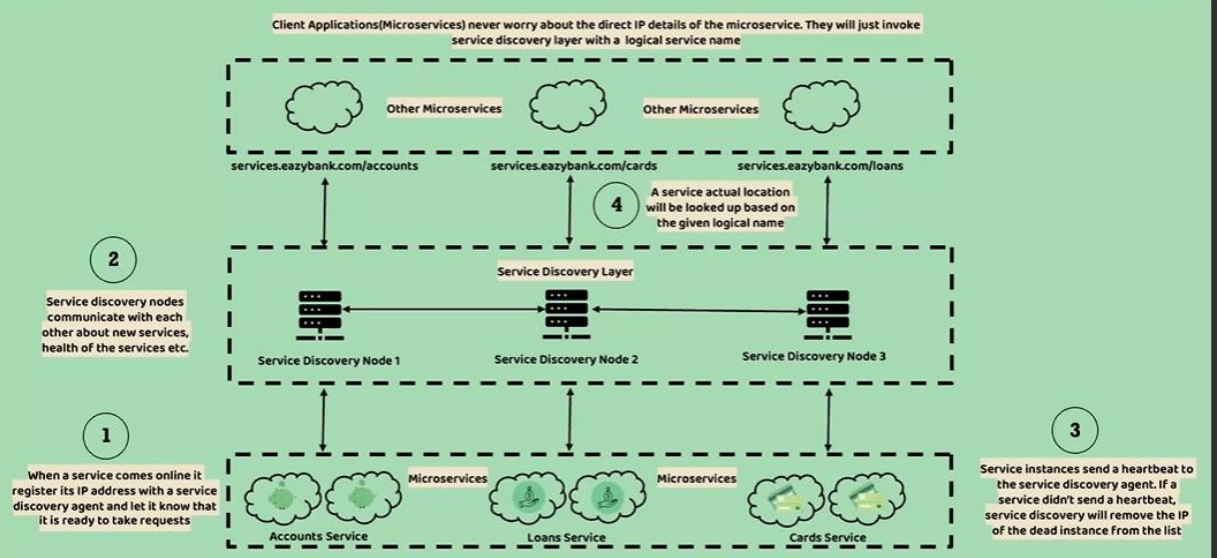
- **Service Registration:** Client applications register themselves with the service registry upon startup. They provide essential information about their location, such as IP address, port, and metadata, which helps identify and categorize the service.
- **Service Discovery:** When a client application needs to communicate with a specific service, it queries the service registry for available instances of that service. The registry responds with the necessary information, such as IP addresses and connection details.
- **Load Balancing:** Client-side service discovery often involves load balancing to distribute the workload across multiple service instances. The client application can implement a load-balancing strategy to select a specific instance based on factors like round-robin, weighted distribution, or latency.

The major advantage of client-side service discovery is load balancing can be implemented using various algorithms, such as round-robin, weighted round-robin, least connections, or even custom algorithms. A drawback is that client service discovery assigns more responsibility to developers. Also, it results in one more service to deploy and maintain (the service registry). Server-side discovery solutions solve these issues. We are going to discuss the same when we are talking about Kubernetes

The Spring Cloud project provides several alternatives for incorporating client-side service discovery in our Spring Boot based microservices. More details to follow...

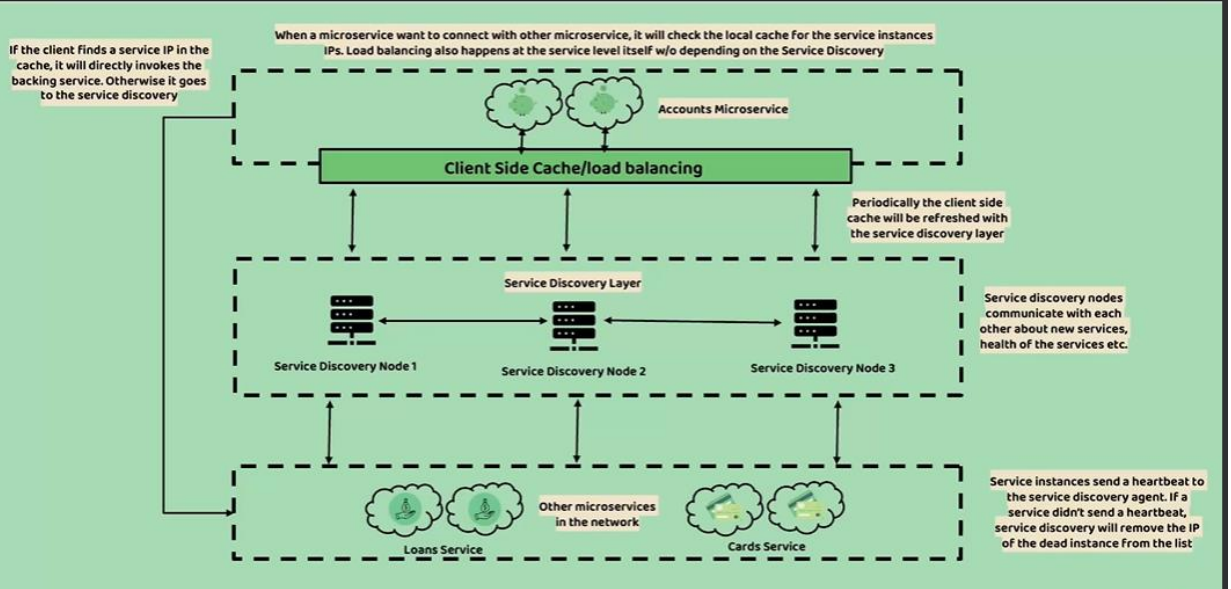
How Client-side service discovery works ?

eazy bytes



How loadbalancing works in Client-side service discovery ?

eazy bytes



Spring Cloud support for Client-side service discovery

eazy bytes

Spring Cloud project makes Service Discovery & Registration setup trivial to undertake with the help of the below components,

- Spring Cloud Netflix's Eureka service which will act as a service discovery agent
- Spring Cloud Load Balancer library for client-side load balancing
- Netflix Feign client to look up for a service b/w microservices

Though in this course we use Eureka since it is mostly used but they are other service registries such as etcd,Consul, and Apache Zookeeper which are also good.

Though NetFlix Ribbon client-side is also good and stable product, we are going to use Spring Cloud Load Balancer for client-side load balancing. This is because Ribbon has entered a maintenance mode and unfortunately, it will not be developed anymore

Steps to build Eureka Server

eazy bytes

Below are the steps to build a Eureka Server application using Spring Cloud Netflix's Eureka,

1. Set up a new Spring Boot project: Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-netflix-eureka-server maven dependency.

2. Configure the properties: In the application properties or YAML file, add the following configurations,

```
server:
 port: 8070

eureka:
 instance:
  hostname: localhost
 client:
  fetchRegistry: false
  registerWithEureka: false
  serviceUrl:
   defaultZone: http://${eureka.instance.hostname}:${server.port}/eureka/
```

3. Add the Eureka Server annotation: In the main class of your project, annotate it with @EnableEurekaServer. This annotation configures the application to act as a Eureka Server.

4. Build and run the Eureka Server: Build your project and run it as a Spring Boot application. Open a web browser and navigate to http://localhost:8070. You should see the Eureka Server dashboard, which displays information about registered service instances.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes

In a distributed system using Eureka, each service instance periodically sends a heartbeat signal to the Eureka server to indicate that it is still alive and functioning. If the Eureka server does not receive a heartbeat from a service instance within a certain timeframe, it assumes that the instance has become unresponsive or has crashed. In normal scenarios, this behavior helps the Eureka server maintain an up-to-date view of the registered service instances.

However, in certain situations, network glitches or temporary system delays may cause the Eureka server to miss a few heartbeats, leading to false expiration of service instances. This can result in unnecessary evictions of healthy service instances from the registry, causing instability and disruption in the system.

To mitigate this issue, Eureka enters into Self-Preservation mode. When Self-Preservation mode is active, the existing registry entries will not be removed even if it stops receiving heartbeats from some of the service instances. This prevents the Eureka server from evicting all the instances due to temporary network glitches or delays.

In Self-Preservation mode, the Eureka server continues to serve the registered instances to client applications, even if it suspects that some instances are no longer available. This helps maintain the stability and availability of the service registry, ensuring that clients can still discover and interact with the available instances.

Self-preservation mode never expires, until and unless the down microservices are brought back or the network glitch is resolved. This is because eureka will not expire the instances till it is above the threshold limit.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes

In a distributed system using Eureka, each service instance periodically sends a heartbeat signal to the Eureka server to indicate that it is still alive and functioning. If the Eureka server does not receive a heartbeat from a service instance within a certain timeframe, it assumes that the instance has become unresponsive or has crashed. In normal scenarios, this behavior helps the Eureka server maintain an up-to-date view of the registered service instances.

However, in certain situations, network glitches or temporary system delays may cause the Eureka server to miss a few heartbeats, leading to false expiration of service instances. This can result in unnecessary evictions of healthy service instances from the registry, causing instability and disruption in the system.

To mitigate this issue, Eureka enters into Self-Preservation mode. When Self-Preservation mode is active, the existing registry entries will not be removed even if it stops receiving heartbeats from some of the service instances. This prevents the Eureka server from evicting all the instances due to temporary network glitches or delays.

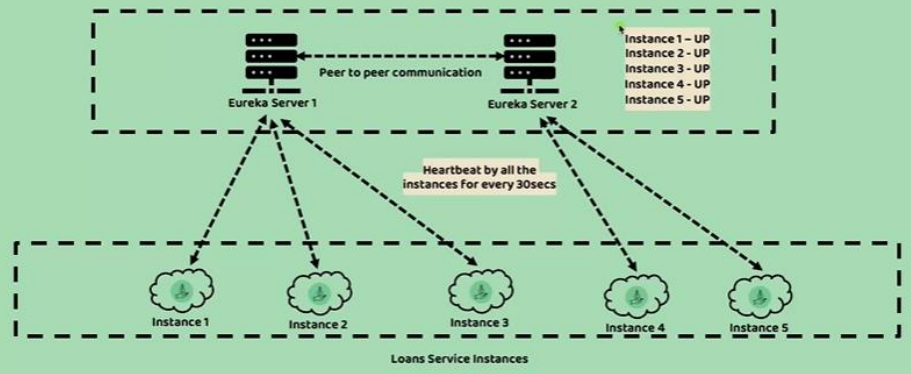
In Self-Preservation mode, the Eureka server continues to serve the registered instances to client applications, even if it suspects that some instances are no longer available. This helps maintain the stability and availability of the service registry, ensuring that clients can still discover and interact with the available instances.

Self-preservation mode never expires, until and unless the down microservices are brought back or the network glitch is resolved. This is because eureka will not expire the instances till it is above the threshold limit.

Eureka Server will not panic when it is not receiving heartbeats from majority of the instances, instead it will be calm and enters into Self-preservation mode. This feature is a savior where the networks glitches are common and help us to handle false-positive alarms.

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

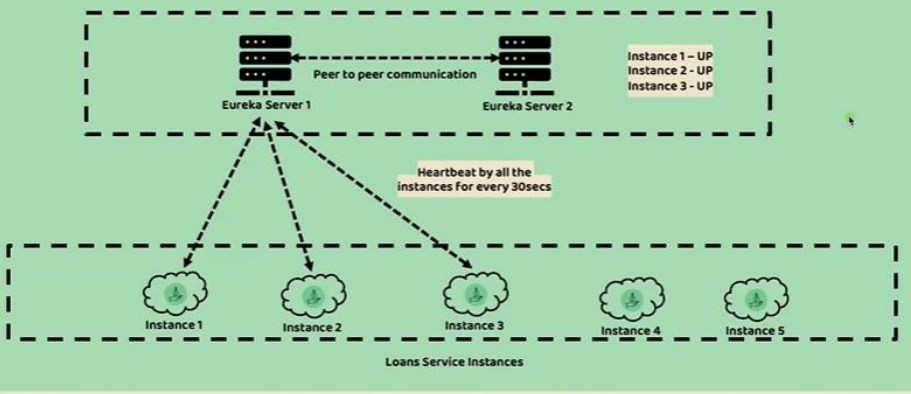
eazy bytes



Healthy Microservices System with all 5 instances up before encountering network problems

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

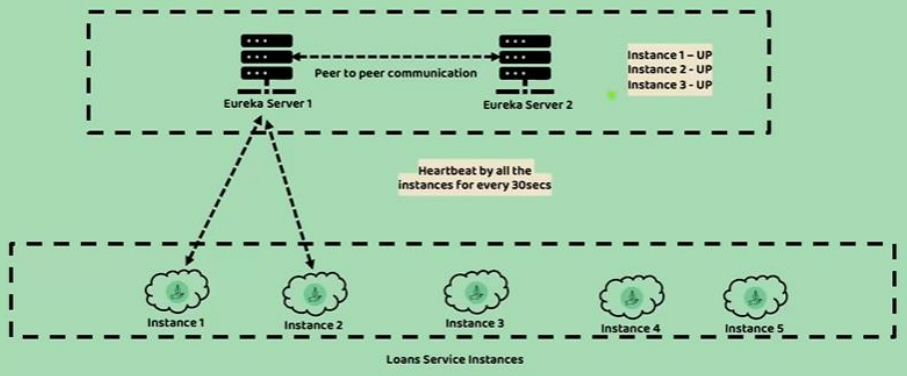
eazy bytes



2 of the instances not sending heartbeat. Eureka enters self-preservation mode since it met threshold percentage

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

eazy bytes



During Self-preservation, eureka will stop expiring the instances though it is not receiving heartbeat from instance 3

EUREKA SELF-PRESERVATION TO AVOID TRAPS IN NETWORK

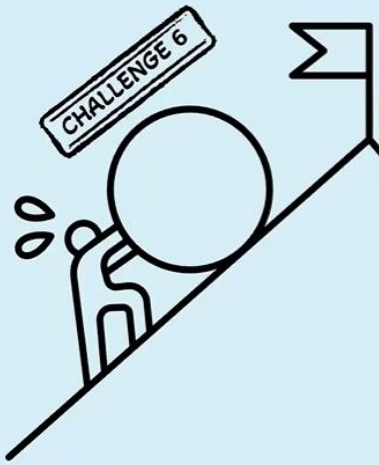
eazy bytes

- Configurations which will directly or indirectly impact self-preservation behavior of eureka

  - ✓ `eureka.instance.lease-renewal-interval-in-seconds = 30`
    Indicates the frequency the client sends heartbeats to server to indicate that it is still alive
  - ✓ `eureka.instance.lease-expiration-duration-in-seconds = 90`
    Indicates the duration the server waits since it received the last heartbeat before it can evict an instance
  - ✓ `eureka.server.eviction-interval-timer-in-ms = 60 * 1000`
    A scheduler(EvictionTask) is run at this frequency which will evict instances from the registry if the lease of the instances are expired as configured by lease-expiration-duration-in-seconds. It will also check whether the system has reached self-preservation mode (by comparing actual and expected heartbeats) before evicting.
  - ✓ `eureka.server.renewal-percent-threshold = 0.85`
    This value is used to calculate the expected % of heartbeats per minute eureka is expecting.
  - ✓ `eureka.server.renewal-threshold-update-interval-ms = 15 * 60 * 1000`
    A scheduler is run at this frequency which calculates the expected heartbeats per minute
  - ✓ `eureka.server.enable-self-preservation = true`
    By default self-preservation mode is enabled but if you need to disable it you can change it to 'false'

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

eazy bytes



HOW DO WE MAINTAIN A SINGLE ENTRYPOINT INTO MICROSERVICES NETWORK ?

How do we build a single gatekeeper for all the inbound traffic to our microservices. This way the client doesn't need to keep track of the different services involved in a transaction, simplifying the client's logic

HOW DO WE HANDLE CROSS CUTTING CONCERNS?

In a distributed microservices architecture, how do we make sure to have a consistently enforced cross cutting concerns like logging, auditing, tracing and security across multiple microservices

HOW DO WE ROUTE BASED ON CUSTOM REQUIREMENTS

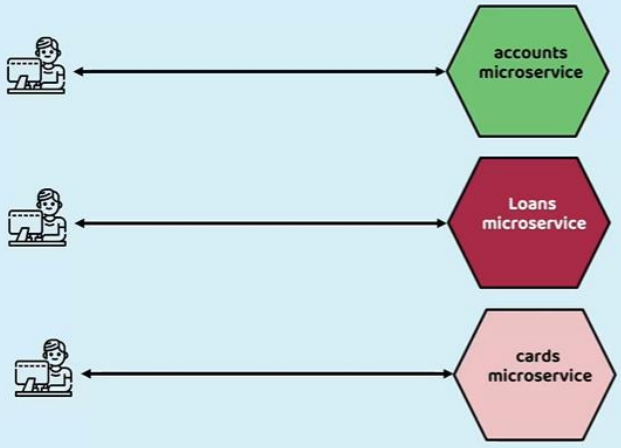
How to provide dynamic routing capabilities which allows to define routing rules based on various criteria such as HTTP headers, request parameters etc. inside microservices network ?

These challenges in microservices can be solved using a

Edge server

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

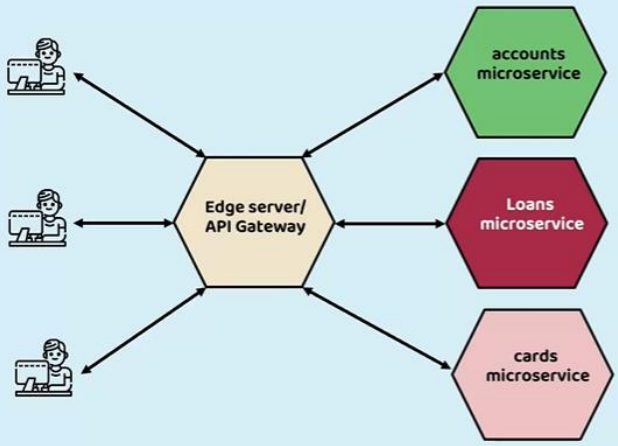
eazy bytes



In a scenario where multiple clients directly connect with various services, several challenges arise. For instance, clients must be aware of the URLs of all the services, and enforcing common requirements such as security, auditing, logging, and routing becomes a repetitive task across all services. To address these challenges, it becomes necessary to establish a single gateway as the entry point to the microservices network.

ROUTING, CROSS CUTTING CONCERNS IN MICROSERVICES

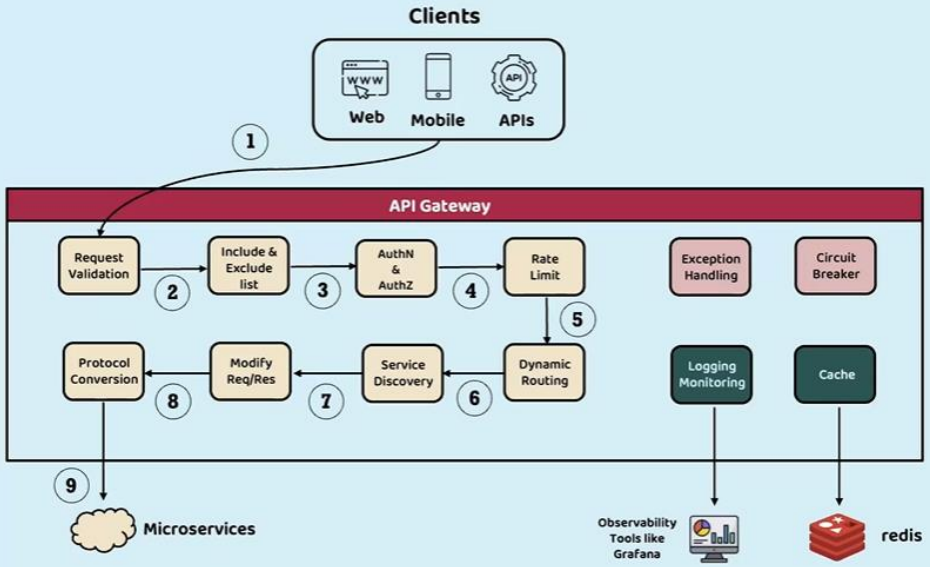
eazy bytes



Edge servers are applications positioned at edge of a system, responsible for implementing functionalities such as API gateways and handling cross-cutting concerns. By utilizing edge servers, it becomes possible to prevent cascading failures when invoking downstream services, allowing for the specification of retries and timeouts for all internal service calls. Additionally, these servers enable control over ingress traffic, empowering the enforcement of quota policies. Furthermore, authentication and authorization mechanisms can be implemented at the edge, enabling the passing of tokens to downstream services for secure communication and access control.

Few important tasks that API Gateway does

eazy bytes



Spring Cloud Gateway

eazy bytes

Spring Cloud Gateway streamlines the creation of edge services by emphasizing ease and efficiency. Moreover, due to its utilization of a reactive framework, it can seamlessly expand to handle the significant workload that typically arises at the system's edge while maintaining optimal scalability.

Here are the key aspects of Spring Cloud Gateway,

- The service gateway sits as the gatekeeper for all inbound traffic to microservice calls within our application. With a service gateway in place, our service clients never directly call the URL of an individual service, but instead place all calls to the service gateway.
- Spring Cloud Gateway is a library for building an API gateway, so it looks like any another Spring Boot application. If you're a Spring developer, you'll find it's very easy to get started with Spring Cloud Gateway with just a few lines of code.
- Spring Cloud Gateway is intended to sit between a requester and a resource that's being requested, where it intercepts, analyzes, and modifies every request. That means you can route requests based on their context. Did a request include a header indicating an API version? We can route that request to the appropriately versioned backend. Does the request require sticky sessions? The gateway can keep track of each user's session.

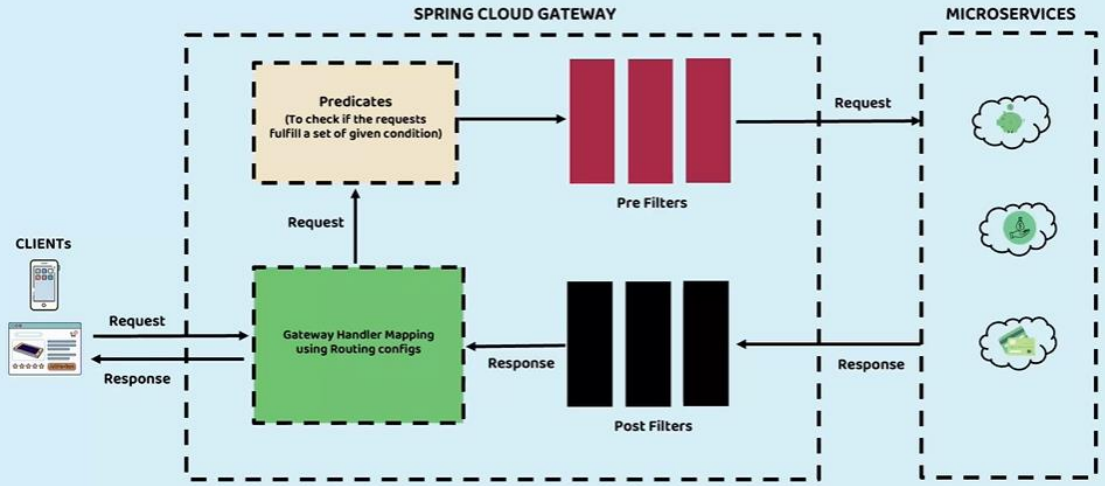
Spring Cloud Gateway is the preferred API gateway compared to zuul. Because Spring Cloud Gateway built on Spring Reactor & Spring WebFlux, provides circuit breaker integration, service discovery with Eureka, non-blocking in nature, has a superior performance compared to that of Zuul.

The service gateway sits between all calls from the client to the individual services & acts as a central Policy Enforcement Point (PEP) like below,

- Routing (Both Static & Dynamic)
- Security (Authentication & Authorization)
- Logging, Auditing and Metrics collection

Spring Cloud Gateway Internal Architecture

eazy bytes



When the client makes a request to the Spring Cloud Gateway, the Gateway Handler Mapping first checks if the request matches a route. This matching is done using the predicates. If it matches the predicate then the request is sent to the pre filters followed by actual microservices. The response will travel through post filters.

Steps to create Spring Cloud Gateway

eazy bytes

Below are the steps to make a microservice application to register and act as a Eureka client,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-gateway, spring-cloud-starter-config & spring-cloud-starter-netflix-eureka-client maven dependencies.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations. Make routing configurations using RouteLocatorBuilder

```yaml
eureka:
 instance:
  preferIpAddress: true
 client:
  registerWithEureka: true
  fetchRegistry: true
  serviceUrl:
   defaultZone: http://localhost:8070/eureka/
spring:
 cloud:
  gateway:
   discovery:
    locator:
     enabled: true
     lowerCaseServiceId: true
```


Steps to create Spring Cloud Gateway

eazy bytes

Below are the steps to make a microservice application to register and act as a Eureka client,

1. **Set up a new Spring Boot project:** Start by creating a new Spring Boot project using your preferred IDE or by using Spring Initializr (https://start.spring.io/). Include the spring-cloud-starter-gateway, spring-cloud-starter-config & spring-cloud-starter-netflix-eureka-client maven dependencies.

2. **Configure the properties:** In the application properties or YAML file, add the following configurations. Make routing configurations using RouteLocatorBuilder

```yaml
eureka:
 instance:
  preferIpAddress: true
 client:
  registerWithEureka: true
  fetchRegistry: true
  serviceUrl:
   defaultZone: http://localhost:8070/eureka/
spring:
 cloud:
  gateway:
   discovery:
    locator:
     enabled: true
     lowerCaseServiceId: true
```

Steps to create Spring Cloud Gateway

eazy bytes

3 **Configure the routing config:** Make routing configurations using RouteLocatorBuilder like shown below,

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
        return builder.routes()
                .route(p -> p
                        .path("/eazybank/accounts/**")
                        .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.*)","/${segment}")
                                .addResponseHeader("X-Response-Time",new Date().toString()))
                        .uri("lb://ACCOUNTS"))
                .route(p -> p
                        .path("/eazybank/loans/**")
                        .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.*)","/${segment}")
                                .addResponseHeader("X-Response-Time",new Date().toString()))
                        .uri("lb://LOANS"))
                .route(p -> p
                        .path("/eazybank/cards/**")
                        .filters(f -> f.rewritePath("/eazybank/cards/(?<segment>.*)","/${segment}")
                                .addResponseHeader("X-Response-Time",new Date().toString()))
                        .uri("lb://CARDS")).build();
}
```

4 **Build and run the application:** Build your project and run it as a Spring Boot application. Invokes the APIs using http://localhost:8072 which is the gateway path.

RESILIENCY IN MICROSERVICES

eazy bytes

HOW DO WE AVOID CASCADING FAILURES?

One failed or slow service should not have a ripple effect on the other microservices. Like in the scenarios of multiple microservices are communicating, we need to make sure that the entire chain of microservices does not fail with the failure of a single microservice

HOW DO WE HANDLE FAILURES GRACEFULLY WITH FALLBACKS?

In a chain of multiple microservices, how do we build a fallback mechanism if one of the microservice is not working. Like returning a default value or return values from cache or call another service/DB to fetch the results etc.

HOW TO MAKE OUR SERVICES SELF-HEALING CAPABLE

In the cases of slow performing services, how do we configure timeouts, retries and give time for a failed services to recover itself.

Ensuring system stability and resilience is crucial for providing a reliable service to users. One of the critical aspects in achieving a stable and resilient system for production is managing the integration points between services over a network.

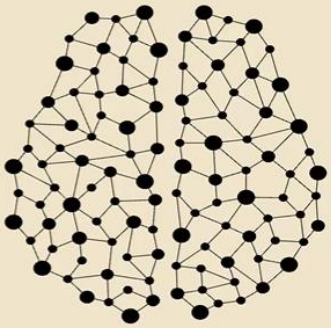
There exist various patterns for building resilient applications. In the Java ecosystem, Hystrix, a library developed by Netflix, was widely used for implementing such patterns. However, Hystrix entered maintenance mode in 2018 and is no longer being actively developed. To address this, Resilience4J has gained significant popularity, stepping in to fill the gap left by Hystrix. Resilience4J provides a comprehensive set of features for building resilient applications and has become a go-to choice for Java developers.

RESILIENCY USING RESILIENCE4J

eazy bytes

Resilience4j is a lightweight fault tolerance library designed for functional programming. It offers the following patterns for increasing fault tolerance due to network problems or failure of any of the multiple services:

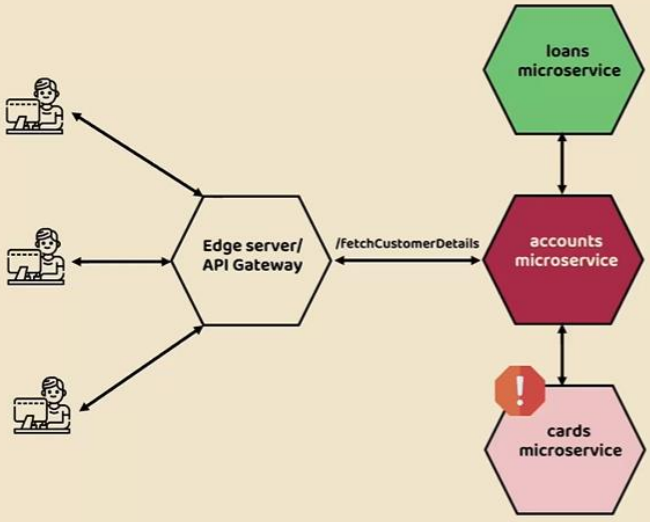
- **Circuit breaker** - Used to stop making requests when a service invoked is failing
- **Fallback** - Alternative paths to failing requests
- **Retry** - Used to make retries when a service has temporarily failed
- **Rate limit** - Limits the number of calls that a service receives in a time
- **Bulkhead** - Limits the number of outgoing concurrent requests to a service to avoid overloading



5x 6:44 / 9:03

TYPICAL SCENARIO IN MICROSERVICES

eazy bytes

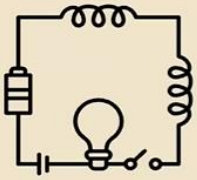


When a microservice responds slowly or fails to function, it can lead to the depletion of resource threads on the Edge server and intermediate services. This, in turn, has a negative impact on the overall performance of the microservice network.

To handle this kind of scenarios, we can use Circuit Breaker pattern

CIRCUIT BREAKER PATTERN

eazy bytes



In an electrical system, a circuit breaker is a safety device designed to protect the electrical circuit from excessive current, preventing damage to the circuit or potential fire hazards. It automatically interrupts the flow of electricity when it detects a fault, such as a short circuit or overload, to ensure the safety and stability of the system.

The Circuit Breaker pattern in software development takes its inspiration from the concept of an electrical circuit breaker found in electrical systems.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them.

The Circuit Breaker pattern which inspired from electrical circuit breaker will monitor the remote calls. If the calls take too long, the circuit breaker will intercede and kill the call. Also, the circuit breaker will monitor all calls to a remote resource, and if enough calls fail, the circuit break implementation will pop, failing fast and preventing future calls to the failing remote resource.

The Circuit Breaker pattern also enables an application to detect whether the fault has been resolved. If the problem appears to have been fixed, the application can try to invoke the operation.

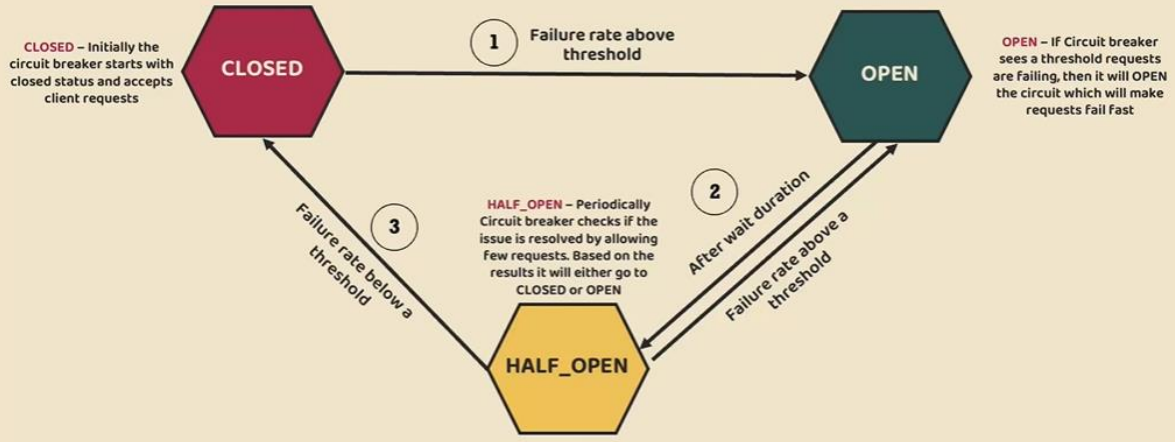
The advantages with circuit breaker pattern are,

- ✓ Fail fast
- ✓ Fail gracefully
- ✓ Recover seamlessly

CIRCUIT BREAKER PATTERN

eazy bytes

In Resilience4j, the circuit breaker is implemented via three states



CIRCUIT BREAKER PATTERN

eazy bytes

Below are the steps to build a circuit breaker pattern using Spring Cloud Gateway Filter,

1. Add maven dependency: Add spring-cloud-starter-circuitbreaker-reactor-resilience4j maven dependency inside pom.xml

2. Add circuit breaker filter: Inside the method where we are creating a bean of RouteLocator, add a filter of circuit breaker like highlighted below and create a REST API handling the fallback uri `/contactSupport`

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
   return builder.routes()
               .route(p -> p.path("/eazybank/accounts/**")
                       .filters(f -> f.rewritePath("/eazybank/accounts/(?<segment>.*)","/${segment}")
                       .addResponseHeader("X-Response-Time",new Date().toString())
                       .circuitBreaker(config -> config.setName("accountsCircuitBreaker")
                       .setFallbackUri("forward:/contactSupport")))
                       .uri("lb://ACCOUNTS")).build();
}
```

3. Add properties: Add the below properties inside the application.yml file,

```
resilience4j.circuitbreaker:
  configs:
    default:
      slidingWindowSize: 10
      permittedNumberOfCallsInHalfOpenState: 2
      failureRateThreshold: 50
      waitDurationInOpenState: 10000
```


CIRCUIT BREAKER PATTERN

eazy bytes

Below are the steps to build a circuit breaker pattern using normal Spring Boot service,

1. Add maven dependency: Add spring-cloud-starter-circuitbreaker-resilience4j maven dependency inside pom.xml

2. Add circuit breaker related changes in Feign Client interfaces like shown below:

```
@FeignClient(name= "cards", fallback = CardsFallback.class)
public interface CardsFeignClient {

   @GetMapping(value = "/api/fetch",consumes = "application/json")
   public ResponseEntity<CardsDto> fetchCardDetails(@RequestHeader("eazybank-correlation-id")
                                   String correlationId, @RequestParam String mobileNumber);

}
```

```
@Component
public class CardsFallback implements CardsFeignClient{
   @Override
   public ResponseEntity<CardsDto> fetchCardDetails(String correlationId, String mobileNumber) {
      return null;
   }
}
```

CIRCUIT BREAKER PATTERN

eazy bytes

3 **Add properties:** Add the below properties inside the application.yml file,

```
spring:
 cloud:
  openfeign:
   circuitbreaker:
    enabled: true
resilience4j.circuitbreaker:
    configs:
      default:
        slidingWindowSize: 5
        failureRateThreshold: 50
        waitDurationInOpenState: 10000
        permittedNumberOfCallsInHalfOpenState: 2
```

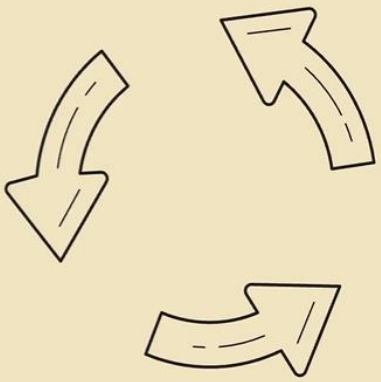

RETRY PATTERN

eazy bytes

The retry pattern will make configured multiple retry attempts when a service has temporarily failed. This pattern is very helpful in the scenarios like network disruption where the client request may successful after a retry attempt.

Here are some key components and considerations of implementing the Retry pattern in microservices:

- **Retry Logic:** Determine when and how many times to retry an operation. This can be based on factors such as error codes, exceptions, or response status.
- **Backoff Strategy:** Define a strategy for delaying retries to avoid overwhelming the system or exacerbating the underlying issue. This strategy can involve gradually increasing the delay between each retry, known as exponential backoff.
- **Circuit Breaker Integration:** Consider combining the Retry pattern with the Circuit Breaker pattern. If a certain number of retries fail consecutively, the circuit can be opened to prevent further attempts and preserve system resources.
- **Idempotent Operations:** Ensure that the retried operation is idempotent, meaning it produces the same result regardless of how many times it is invoked. This prevents unintended side effects or duplicate operations.



RETRY PATTERN

eazy bytes

Below are the steps to build a retry pattern using Spring Cloud Gateway filter,

1 **Add Retry Filter:** Inside the method where we are creating a bean of RouteLocator, add a filter of retry like highlighted below,

```
@Bean
public RouteLocator myRoutes(RouteLocatorBuilder builder) {
  return builder.routes()
      .route(p -> p.path("/eazybank/loans/**")
          .filters(f -> f.rewritePath("/eazybank/loans/(?<segment>.*)","/${segment}")
              .addResponseHeader("X-Response-Time",new Date().toString())
              .retry(retryConfig -> retryConfig.setRetries(3).setMethods(HttpMethod.GET)
                  .setBackoff(Duration.ofMillis(100),Duration.ofMillis(1000),2,true)))
          .uri("lb://LOANS")).build();
}
```


OBSERVABILITY AND MONITORING OF MICROSERVICES

eazy bytes

DEBUGGING A PROBLEM IN MICROSERVICES ?

How do we trace transactions across multiple services, containers and try to find where exactly the problem or bug is?

How do we combine all the logs from multiple services into a central location where they can be indexed, searched, filtered, and grouped to find bugs that are contributing to a problem?

MONITORING PERFORMANCE OF SERVICE CALLS?

How can we track the path of a specific chain service call through our microservices network, and see how long it took to complete at each microservice?

MONITORING SERVICES METRICS & HEALTH ?

How can we easily and efficiently monitor the metrics like CPU usage, JVM metrics, etc. for all the microservices applications in our network?

How can we monitor the status and health of all of our microservices applications in a single place, and create alerts and notifications for any abnormal behavior of the services?

Observability and **monitoring** solve the challenge of identifying and resolving above problems in microservices architectures before they cause outages.

WHAT IS OBSERVABILITY ?

eazy bytes

Observability is the ability to understand the internal state of a system by observing its outputs. In the context of microservices, observability is achieved by collecting and analyzing data from a variety of sources, such as metrics, logs, and traces.

The three pillars of observability are:

- **Metrics:** Metrics are quantitative measurements of the health of a system. They can be used to track things like CPU usage, memory usage, and response times.
- **Logs:** Logs are a record of events that occur in a system. They can be used to track things like errors, exceptions, and other unexpected events.
- **Traces:** Traces are a record of the path that a request takes through a system. They can be used to track the performance of a request and to identify bottlenecks.

By collecting and analyzing data from these three sources, you can gain a comprehensive understanding of the internal state of your microservices architecture. This understanding can be used to identify and troubleshoot problems, improve performance, and ensure the overall health of your system.

WHAT IS MONITORING ?

eazy bytes

Monitoring in microservices involves checking the telemetry data available for the application and defining alerts for known failure states. This process collects and analyzes data from a system to identify and troubleshoot problems, as well as track the health of individual microservices and the overall health of the microservices network.

Monitoring in microservices is important because it allows you to:

- **Identify and troubleshoot problems:** By collecting and analyzing data from your microservices, you can identify problems before they cause outages or other disruptions.
- **Track the health of your microservices:** Monitoring can help you to track the health of your microservices, so you can identify any microservices that are underperforming or that are experiencing problems.
- **Optimize your microservices:** By monitoring your microservices, you can identify areas where you can optimize your microservices to improve performance and reliability.

Monitoring and observability can be considered as two sides of the same coin. Both rely on the same types of telemetry data to enable insight into software distributed systems. Those data types — metrics, traces, and logs — are often referred to as the three pillars of observability.

Observability vs. Monitoring

eazy bytes

Feature	Monitoring	Observability
Purpose	Identify and troubleshoot problems	Understand the internal state of a system
Data	Metrics, traces, and logs	Metrics, traces, logs, and other data sources
Goal	Identify problems	Understand how a system works
Approach	Reactive	Proactive

In other words, monitoring is about collecting data and observability is about understanding data.

Monitory is reacting to problems while observavility is fixing them in real time.

Logging

eazy bytes

Logs are discrete records of events that happen in software applications over time. They contain a timestamp that indicates when the event happened, as well as information about the event and its context. This information can be used to answer questions like "What happened at this time?", "Which thread was processing the event?", or "Which user/tenant was in the context?"

Logs are essential tools for troubleshooting and debugging tasks. They can be used to reconstruct what happened at a specific point in time in a single application instance. Logs are typically categorized according to the type or severity of the event, such as trace, debug, info, warn, and error. This allows us to log only the most severe events in production, while still giving us the chance to change the log level temporarily during debugging.

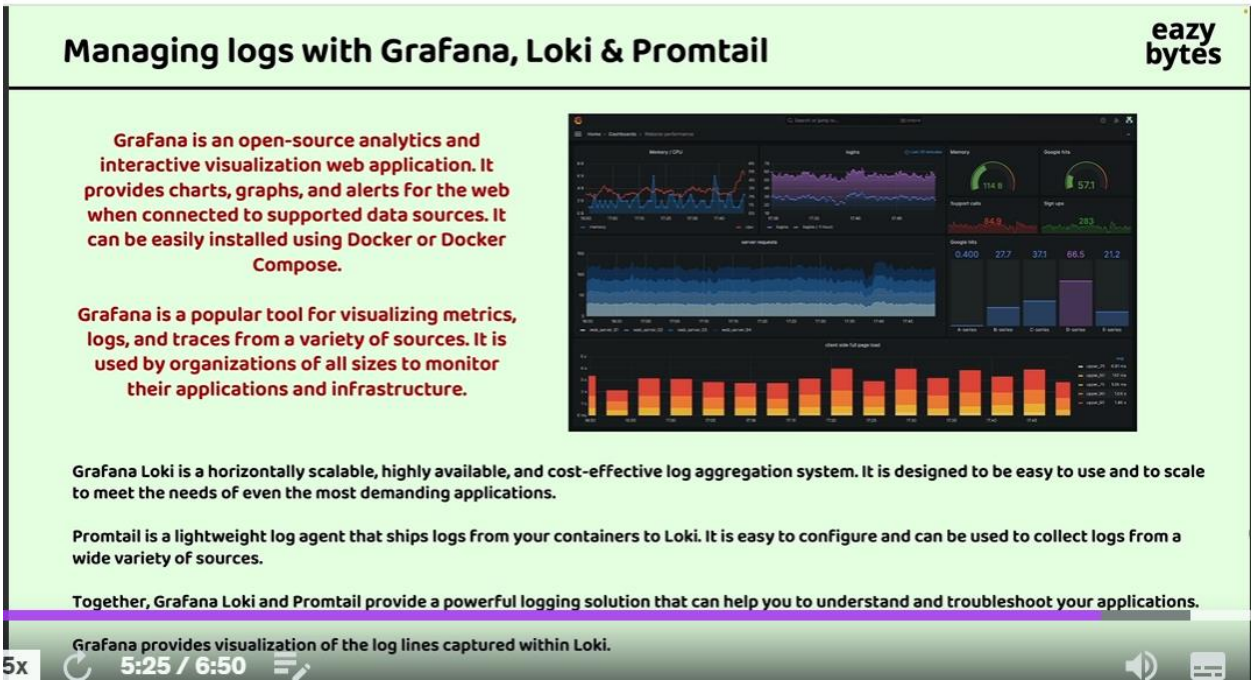
Logging in Monolithic Apps

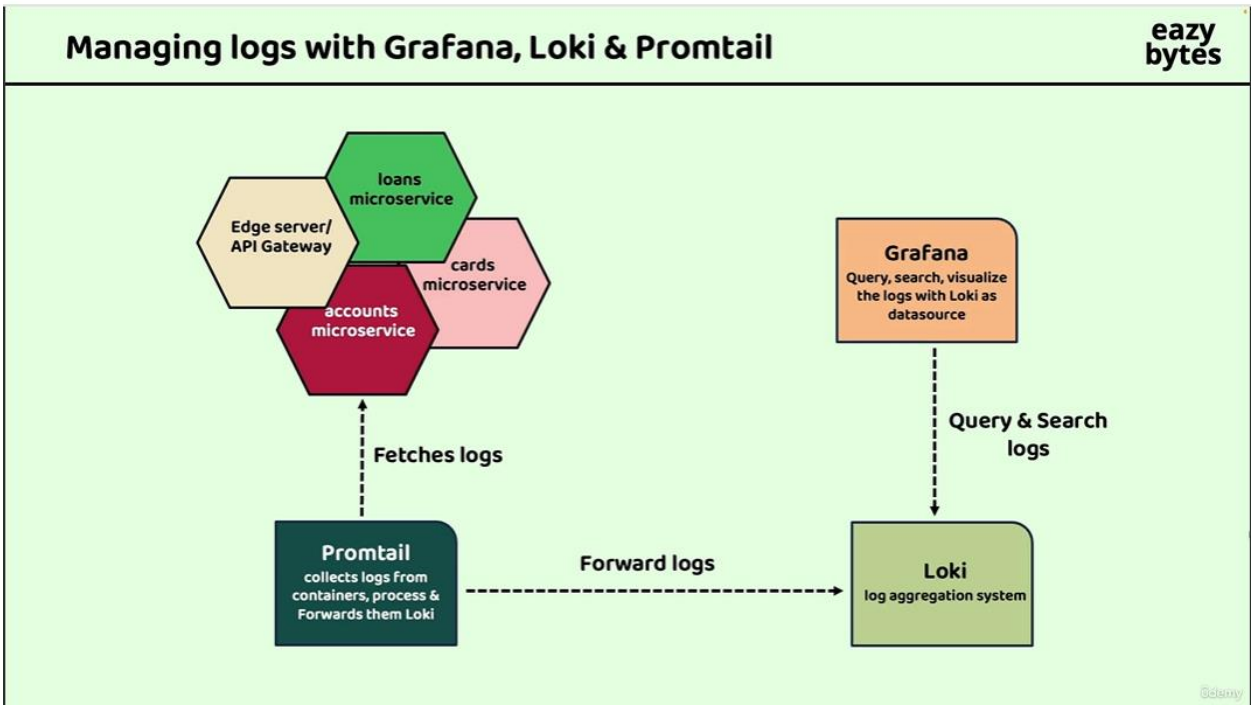
In monolithic apps, all of the code is in a single codebase. This means that all of the logs are also in a single location. This makes it easy to find and troubleshoot problems, as you only need to look in one place.

Logging in Microservices

Logging in microservices is complex. This is because each service has its own logs. This means that you need to look in multiple places to find all of the logs for a particular request.

To address this challenge, microservices architectures often use centralized logging. Centralized logging collects logs from all of the services in the architecture and stores them in a single location. This makes it easier to find and troubleshoot problems, as you only need to look in one place





IMPORTANT NOTE ON PROMTAIL

👉 PROMTAIL IS REPLACED BY ALLOY

1. From **Grafana Loki version 3.0** onwards, **Promtail**, which is responsible for scraping log lines, has been replaced with a new product called **Alloy**. Even though I will discuss Promtail in the next few lectures, Alloy will function

similarly. Since these are internal components of Grafana Loki, this change will not have a significant impact. We just need to use the config files related to Alloy in place of Promtail.

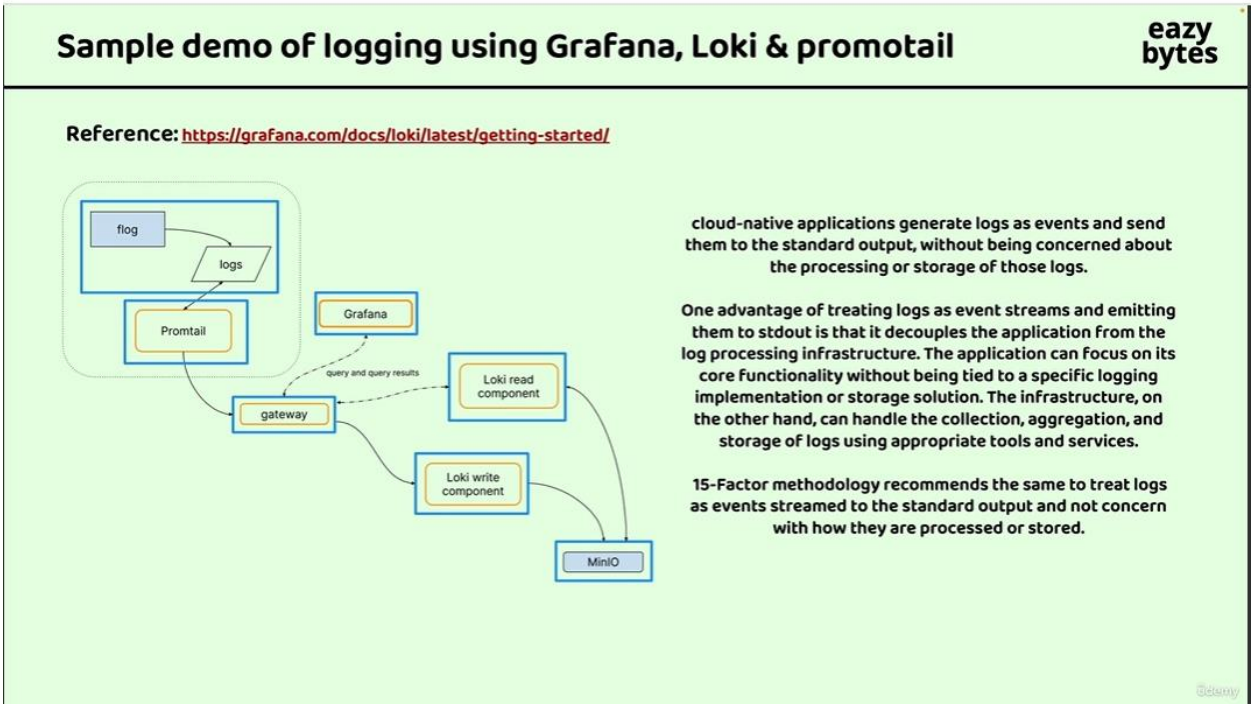
2. All the Docker Compose files have been updated with Alloy-related changes inside the GitHub repo. You can find the documentation for Alloy along with Loki in the following link:

   https://grafana.com/docs/loki/latest/get-started/quick-start/

3. If for any reason your real projects use older versions of Grafana Loki, then you will need to use Promtail. Promtail-related changes are available in the older branches of the course GitHub repo.

Thanks much. Let's have fun and learning,

Madan

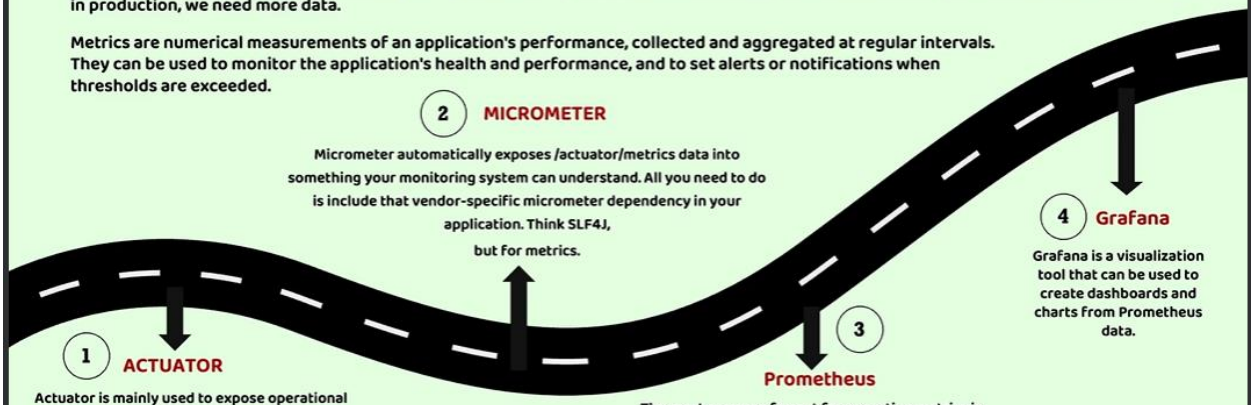


Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana

eazy bytes

Event logs are essential for monitoring applications, but they don't provide enough data to answer all of the questions we need to know. To answer questions like CPU usage, memory usage, threads usage, error requests etc. & properly monitor, manage, and troubleshoot an application in production, we need more data.

Metrics are numerical measurements of an application's performance, collected and aggregated at regular intervals. They can be used to monitor the application's health and performance, and to set alerts or notifications when thresholds are exceeded.



2 MICROMETER

Micrometer automatically exposes /actuator/metrics data into something your monitoring system can understand. All you need to do is include that vendor-specific micrometer dependency in your application. Think SLF4J, but for metrics.

4 Grafana

Grafana is a visualization tool that can be used to create dashboards and charts from Prometheus data.

1 ACTUATOR

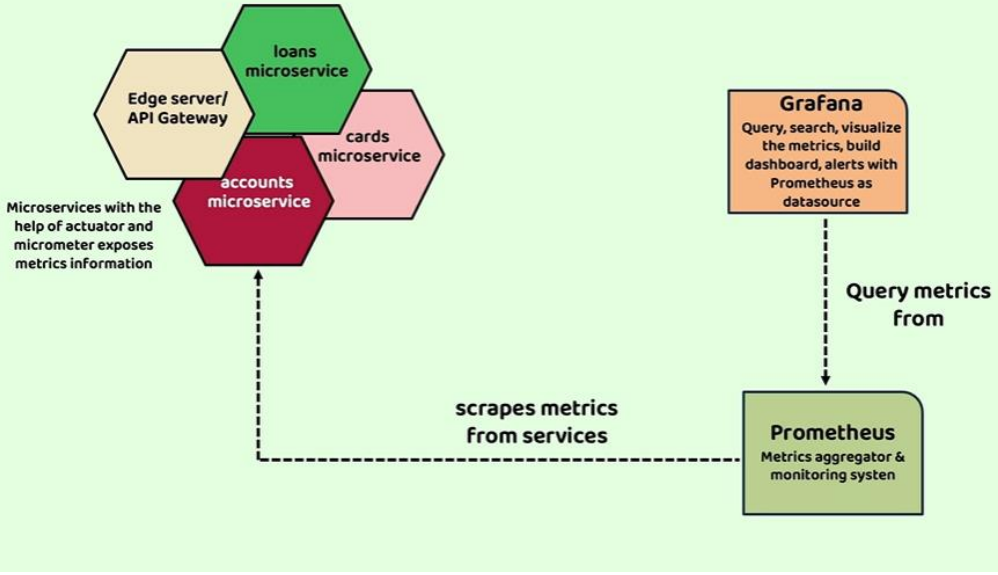
Actuator is mainly used to expose operational information about the running application — health, metrics, info, dump, env, etc. It uses HTTP endpoints or JMX beans to enable us to interact with it.

3 Prometheus

The most common format for exporting metrics is the one used by Prometheus, which is "an open-source systems monitoring and alerting toolkit". Just as Loki aggregates and stores event logs, Prometheus does the same with metrics.

Metrics & monitoring with Spring Boot Actuator, Micrometer, Prometheus & Grafana

eazy bytes



Distributed tracing in microservices

eazy bytes

Event logs, health probes, and metrics offer a wealth of valuable information for deducing the internal condition of an application. Nevertheless, these sources fail to account for the distributed nature of cloud-native applications. Given that a user request often traverses multiple applications, we currently lack the means to effectively correlate data across application boundaries.

Distributed tracing is a technique used in microservices or cloud-native applications to understand and analyze the flow of requests as they propagate across multiple services and components. It helps in gaining insights into how requests are processed, identifying performance bottlenecks, and diagnosing issues in complex, distributed systems.

One possible solution to address this issue is to implement a straightforward approach where a unique identifier, known as a correlation ID, is generated for each request at the entry point of the system. This correlation ID can then be utilized in event logs and passed along to other relevant services involved in processing the request. By leveraging this correlation ID, we can retrieve all log messages associated with a specific transaction from multiple applications.

Distributed tracing encompasses three primary concepts:

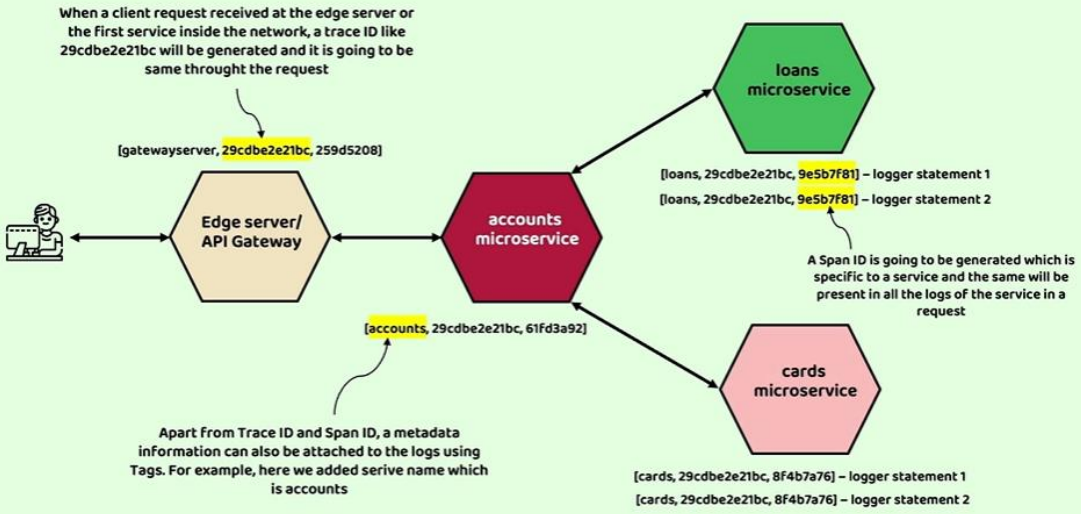
Tags serve as metadata that offer supplementary details about the span context, including the request URI, the username of the authenticated user, or the identifier for a specific tenant.

A trace denotes the collection of actions tied to a request or transaction, distinguished by a trace ID. It consists of multiple spans that span across various services.

A span represents each individual stage of request processing, encompassing start and end timestamps, and is uniquely identified by the combination of trace ID and span ID.

Distributed tracing in microservices

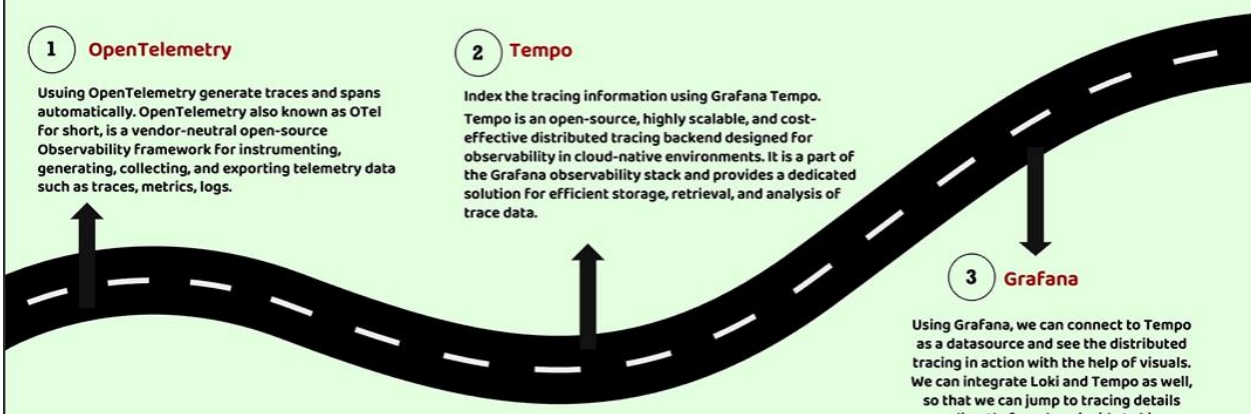
eazy bytes



Udemy

Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes



1 OpenTelemetry

Usuing OpenTelemetry generate traces and spans automatically. OpenTelemetry also known as OTel for short, is a vendor-neutral open-source Observability framework for instrumenting, generating, collecting, and exporting telemetry data such as traces, metrics, logs.

2 Tempo

Index the tracing information using Grafana Tempo.

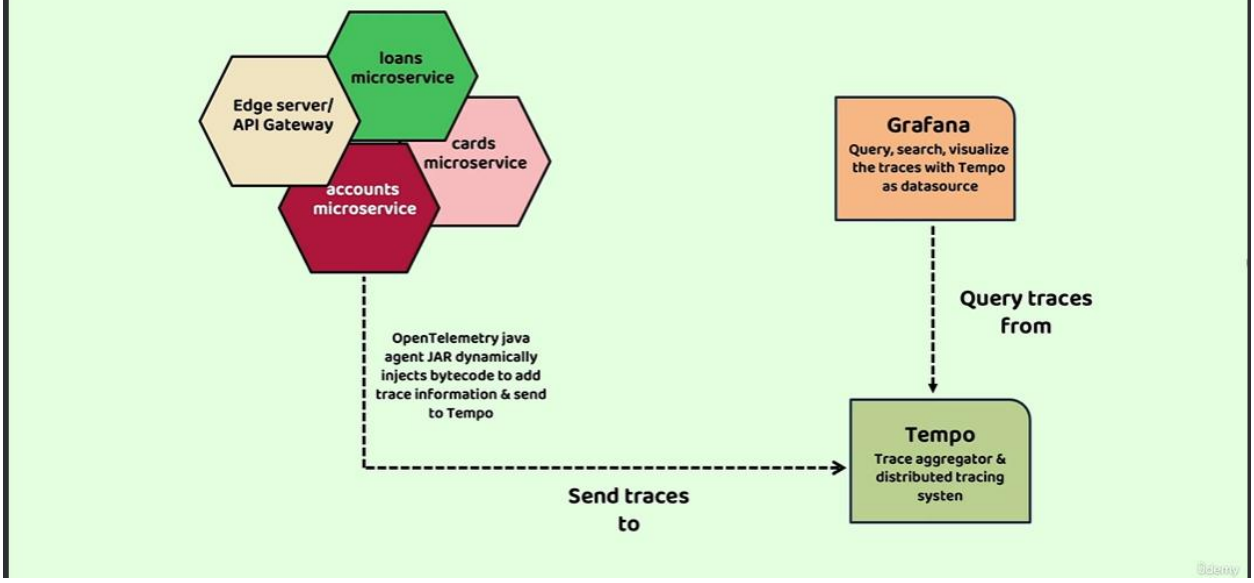
Tempo is an open-source, highly scalable, and cost-effective distributed tracing backend designed for observability in cloud-native environments. It is a part of the Grafana observability stack and provides a dedicated solution for efficient storage, retrieval, and analysis of trace data.

3 Grafana

Using Grafana, we can connect to Tempo as a datasource and see the distributed tracing in action with the help of visuals. We can integrate Loki and Tempo as well, so that we can jump to tracing details directly from logs inside Loki

Distributed tracing with OpenTelemetry, Tempo & Grafana

eazy bytes



Logging

eazy bytes

Logs are discrete records of events that happen in software applications over time. They contain a timestamp that indicates when the event happened, as well as information about the event and its context. This information can be used to answer questions like "What happened at this time?", "Which thread was processing the event?", or "Which user/tenant was in the context?"

Logs are essential tools for troubleshooting and debugging tasks. They can be used to reconstruct what happened at a specific point in time in a single application instance. Logs are typically categorized according to the type or severity of the event, such as trace, debug, info, warn, and error. This allows us to log only the most severe events in production, while still giving us the chance to change the log level temporarily during debugging.

Logging in Monolithic Apps

In monolithic apps, all of the code is in a single codebase. This means that all of the logs are also in a single location. This makes it easy to find and troubleshoot problems, as you only need to look in one place.

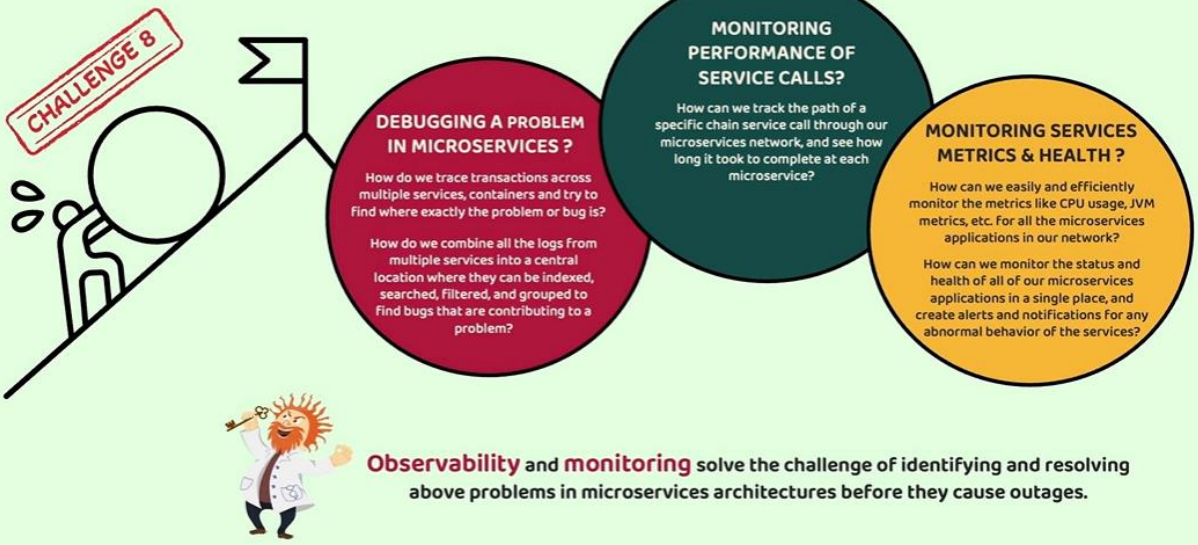
Logging in Microservices

Logging in microservices is complex. This is because each service has its own logs. This means that you need to look in multiple places to find all of the logs for a particular request.

To address this challenge, microservices architectures often use centralized logging. Centralized logging collects logs from all of the services in the architecture and stores them in a single location. This makes it easier to find and troubleshoot problems, as you only need to look in one place

OBSERVABILITY AND MONITORING OF MICROSERVICES

eazy bytes



Observability and **monitoring** solve the challenge of identifying and resolving above problems in microservices architectures before they cause outages.

MICROSERVICES SECURITY

eazy bytes

SECURING MICROSERVICES FROM UNAUTHORIZED ACESS ?

How to secure our microservices and protect them from unauthorized access by client applications or end users ? Right now all our services doesn't have security and any one can invoke them to get a reponse which have sensitive data

AUTHENTICATION AND AUTHORIZATION

How can our microservices can authenticate and authorize users and services to access them. Our microservices should be capable of performing identification, authentication & authorization.

CENTRALIZED IDENTITY AND ACCESS MANAGEMENT (IAM)

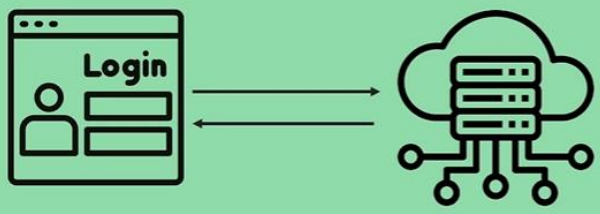
How to maintain a centralized component to store user credentials, handling identity & access management

Using **OAuth2/OpenID Connect, KeyCloak (IAM), Spring Security** we can secure the microservices and handle all the above challenges.

PROBLEM THAT OAUTH2 SOLVES

eazy bytes

Why should we use OAUTH2 framework for implementing security inside our microservices ? Why can't we use the basic authentication ? To answer this, first lets try to understand the basic authentication & it's drawbacks.



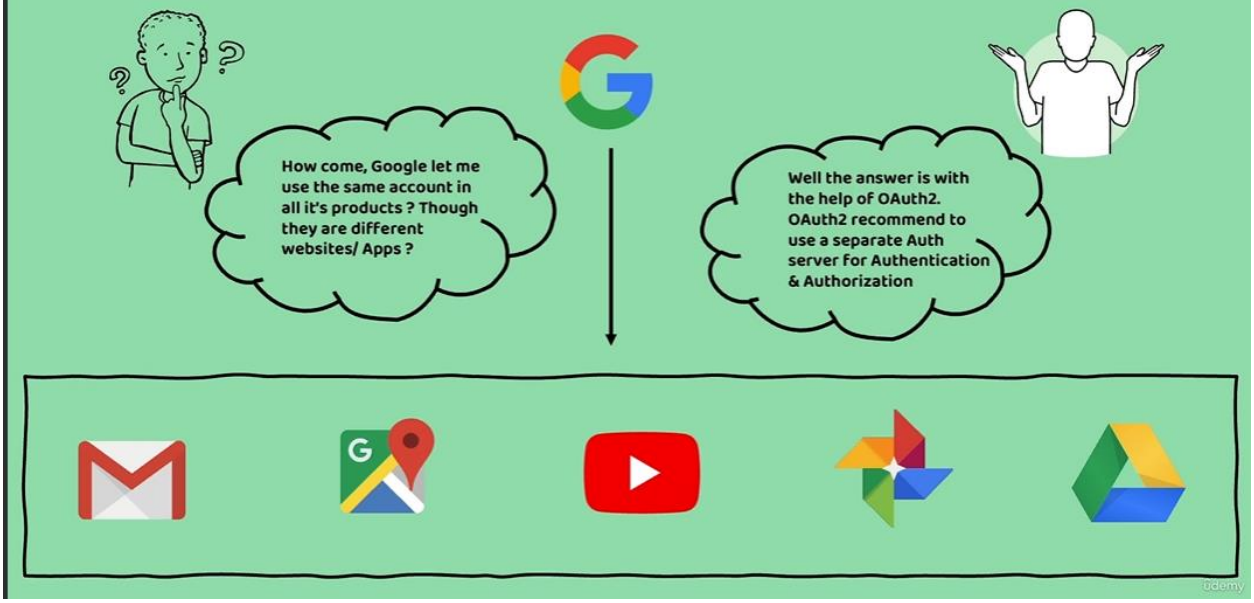
Early websites usually ask for credentials via an HTML form, which the browser will send to the server. The server authenticates the information and writes a session value in the cookie; as long as the session is still marked active, user can access protected features and resources.

Drawbacks of Basic authentication

- Backend server or business logic is tightly coupled with the Authentication/Authorization logic. Not mobile flow/ REST APIs friendly
- Basic authentication flow does not accommodate well the use case where users of one product or service would like to grant third-party clients access to their information on the platform.

PROBLEM THAT OAUTH2 SOLVES

eazy bytes



INTRODUCTION TO OAUTH2

eazy bytes

OAuth stands for Open Authorization. It's a free and open protocol, built on IETF standards and licenses from the Open Web Foundation. OAuth 2.1 is a security standard where you give one application permission to access your data in another application. The steps to grant permission, or consent, are often referred to as authorization or even delegated authorization. You authorize one application to access your data, or use features in another application on your behalf, without giving them your password.

Below are the few advantages of OAuth2 standard,

- **Supports all kinds of Apps:** OAuth2 supports multiple use cases addressing different device capabilities. It supports server-to-server apps, browser-based apps, mobile/native apps, IoT devices and consoles/TVs. It has various authorization grant flows like Authorization Code grant, Client Credentials Grant Type etc. to support all kinds of apps communication.

- **Separation of Auth logic:** Inside OAuth2, we have Authorization Server which receives requests from the Client for Access Tokens and issues them upon successful authentication. This enable us to maintain all the security logic in a single place. Regardless of how many applications an organization has, they all can connect to Auth server to perform login operation.

  All user credentials & client application credentials will be maintained in a single location which is inside Auth Server.

- **No need to share Credentials:** If you plan to allow a third-party applications and services to access your resources, then there is no need to share your credentials.

  In many ways, you can think of the OAuth2 token as a "access card" at any office/hotel. These tokens provides limited access to someone, without handing over full control in the form of the master key.

OAUTH2 TERMINOLOGY

eazy bytes

Resource owner – It is you the end user. In the scenario of Stackoverflow, the end user who want to use the GitHub services to get his details. In other words, the end user owns the resources (email, profile), that's why we call him as Resource owner

Client – The website, mobile app or API will be the client as it is the one which interacts with GitHub services on behalf of the resource owner/end user. In the scenario of Stackoverflow, the Stackoverflow website is Client

Authorization Server – This is the server which knows about resource owner. In other words, resource owner should have an account in this server. In the scenario of Stackoverflow, the GitHub server which has authorization logic acts as Authorization server.

Resource Server – This is the server where the resources that client want to consume are hosted. In the scenario of Stackoverflow, the resources like User Email, Profile details are hosted inside GitHub server. So it will act as a resource server.

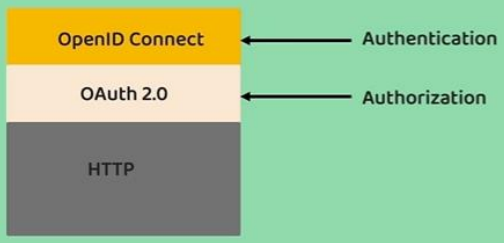
Scopes – These are the granular permissions the Client wants, such as access to data or to perform certain actions. The Auth server can issue an access token to client with the scope of Email, READ etc.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

What is OpenID Connect?

- OpenID Connect is a protocol that sits on top of the OAuth 2.0 framework. While OAuth 2.0 provides authorization via an access token containing scopes, OpenID Connect provides authentication by introducing a new ID token which contains a new set of information and claims specifically for identity.
- With the ID token, OpenID Connect brings standards around sharing identity details among the applications.



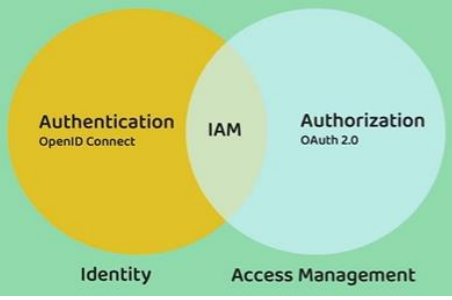
The OpenID Connect flow looks the same as OAuth. The only differences are, in the initial request, a specific scope of openid is used, and in the final exchange the client receives both an Access Token and an ID Token.

WHAT IS OPENID CONNECT & WHY IT IS IMPORTANT ?

eazy bytes

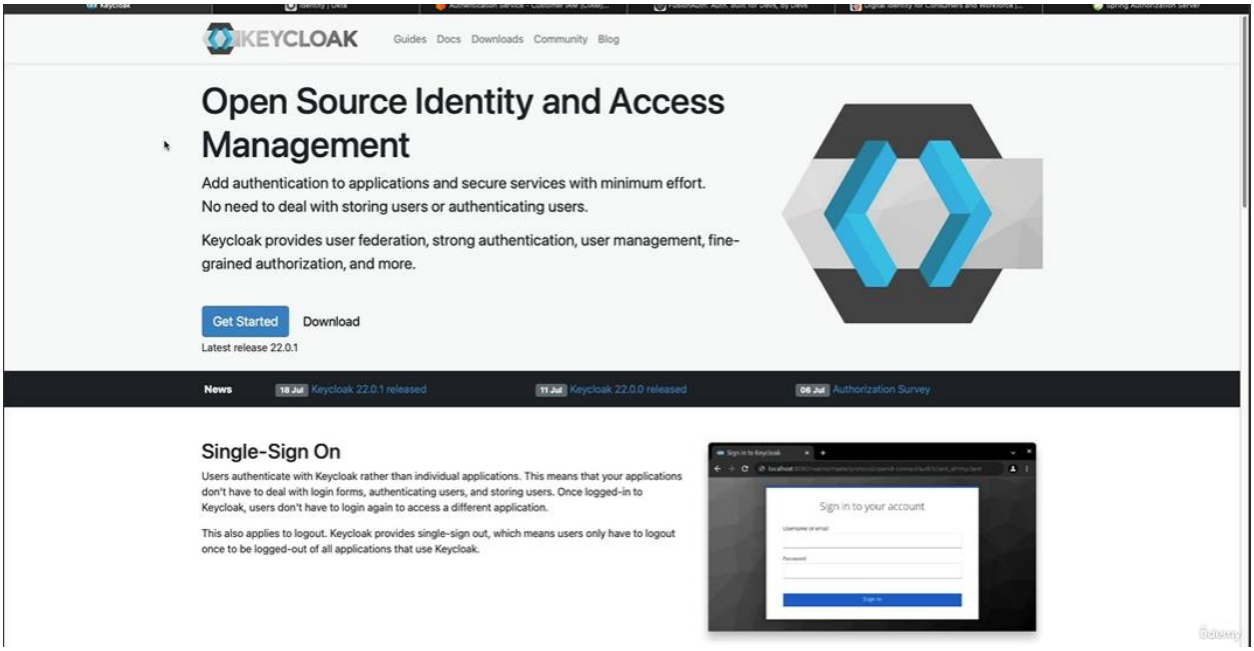
Why is OpenID Connect important?

- Identity is the key to any application. At the core of modern authorization is OAuth 2.0, but OAuth 2.0 lacks an authentication component. Implementing OpenID Connect on top of OAuth 2.0 completes an IAM (Identity & Access Management) strategy.
- As more and more applications need to connect with each other and more identities are being populated on the internet, the demand to be able to share these identities is also increased. With OpenID connect, applications can share the identities easily and standard way.



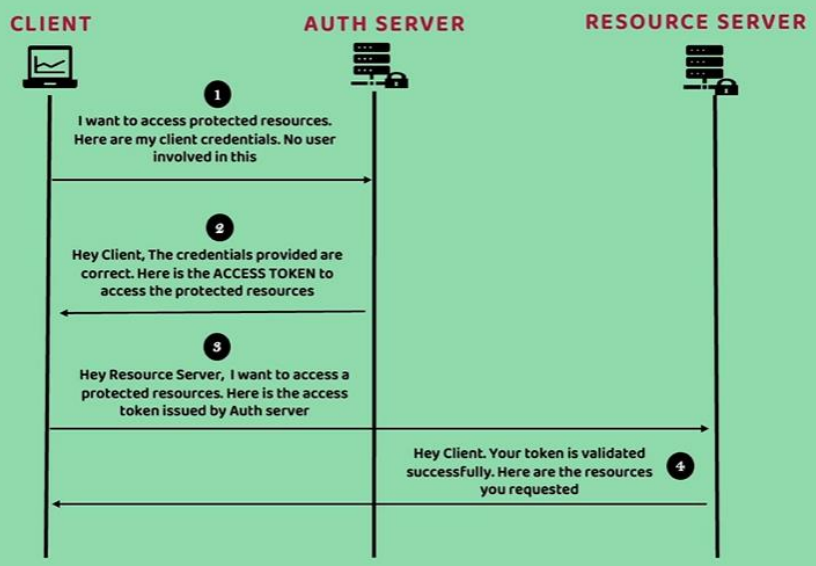
OpenID Connect add below details to OAuth 2.0

1. OIDC standardizes the scopes to openid, profile, email, and address.
2. ID Token using JWT standard
3. OIDC exposes the standardized "/userinfo" endpoint.



CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

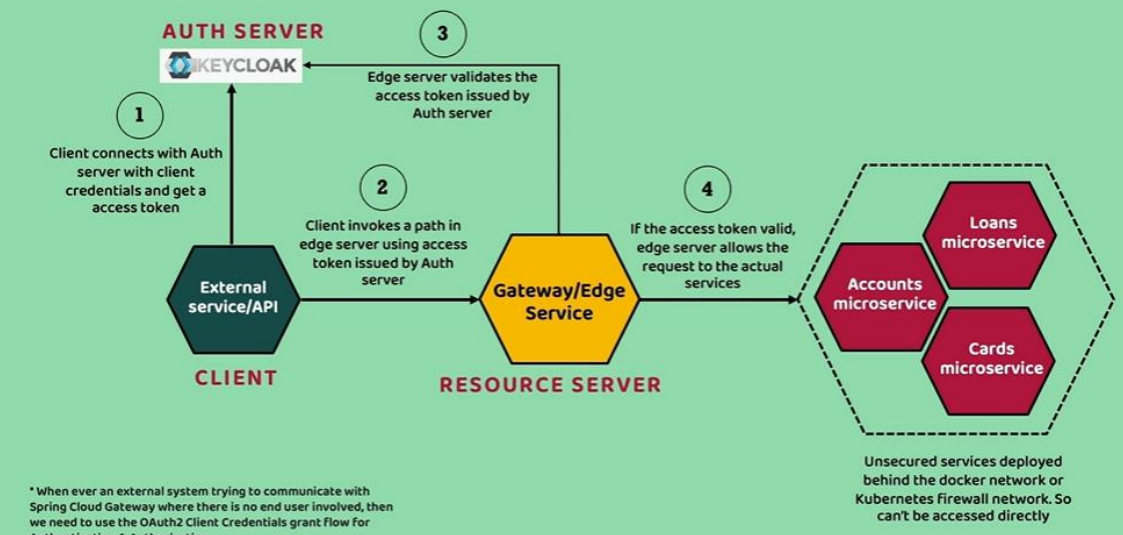
eazy bytes

- ✓ In the step 1, where client is making a request to Auth Server endpoint, have to send the below important details,
  - **client_id & client_secret** – the credentials of the client to authenticate itself.
  - **scope** – similar to authorities. Specifies level of access that client is requesting like EMAIL, PROFILE
  - **grant_type** – With the value 'client_credentials' which indicates that we want to follow client credentials grant type

- ✓ This is the most simplest grant type flow in OAUTH2.
- ✓ We use this authentication flow only if there is no user and UI involved. Like in the scenarios where 2 different applications want to share data between them using backend APIs.

SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes

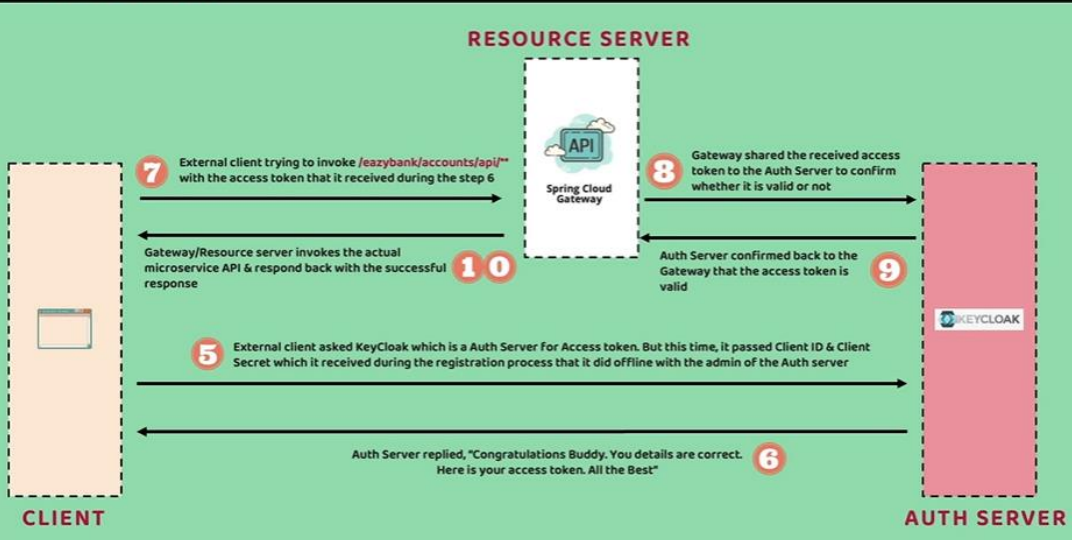


* When ever an external system trying to communicate with Spring Cloud Gateway where there is no end user involved, then we need to use the OAuth2 Client Credentials grant flow for Authentication & Authorization.

Udemy

SECURING GATEWAY USING CLIENT CREDENTIALS GRANT TYPE FLOW IN OAUTH2

eazy bytes



Udemy